Evaluating Meta-Programming Mechanisms for ORB Middleware

Nanbor Wang and Kirthika Parameswaran
{nanbor, kirthika}@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis

Douglas Schmidt and Ossama Othman
{schmidt, ossama}@uci.edu
Electrical & Computer Engineering
University of California, Irvine

Abstract

Distributed object computing (DOC) middleware, such as CORBA, COM+, and Java RMI, shields developers from many tedious and error-prone aspects of programming distributed applications. It is hard to evolve distributed applications after they are deployed, however, without adequate middleware support for meta-programming mechanisms, such as smart proxies, interceptors, and pluggable protocols. These mechanisms can help improve the adaptability of distributed applications by allowing their behavior to be modified without changing their existing software designs and implementations significantly.

This article examines and compares common meta-programming mechanisms supported by DOC middleware. These mechanisms allow applications to adapt more readily to changes in requirements and run-time environments throughout their lifecycles. Some of these meta-programming mechanisms are relatively new, whereas others have existed for decades. Until recently, however, DOC middleware has not provided all these mechanisms in a single integrated framework, so researchers and developers may not be familiar with the breadth of meta-programming mechanisms available today. This article provides a systematic evaluation of these mechanisms to help researchers and developers determine which are best suited for their application needs.

1 Introduction

Motivation: Developers of distributed applications face many challenges stemming from inherent and accidental complexities, such as latency, partial failure, and non-portable low-level OS APIs. The magnitude of these complexities—combined with increasing time-to-market pressures—make it increasingly impractical to develop complex distributed applications manually from scratch. Commercial-off-the-shelf (COTS) distributed object computing (DOC) middleware helps address these challenges by:

1. Defining standard higher-level programming abstractions, such as distributed object and component interfaces, that provide location and platform transparency to client and server components;

2. Shielding application developers from low-level network programming details, such as connection management, data transfer, parameter (de)marshaling, endpoint and request demultiplexing, error handling, multi-threading synchronization, and fault tolerance; and

3. Amortizing software lifecycle costs by leveraging previous development expertise and reifying the implementation and deployment of key patterns [1, 2] via reusable middleware frameworks and common services.

In the case of standards-based DOC middleware, such as Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [3], these capabilities are realized through an open specification process. The resulting products can therefore interoperate across many OS/network/hardware platforms and programming languages [4].

To date, DOC middleware has been used successfully in domains ranging from telecommunications to aerospace, process automation, and e-commerce. It has enabled developers to create applications rapidly that can meet a particular set of requirements with a reasonable amount of effort. DOC middleware has been less successful, however, at shielding developers from the effects of changes to requirements or environmental conditions that occur late in an application's lifecycle, *i.e.*, during deployment and/or at run-time. For example, application developers may want to change the following aspects of their programs after initial deployment:

- Adding security mechanisms, such as authentication of credentials
- Buffering invocations to enable batch transfer and minimize calls to remote target objects
- Supporting advanced quality-of-service (QoS) features, such as multi-level distributed resource management based on adaptive feedback control
- Configuring new transport protocols that are tailored for a particular network environment
- Monitoring application behavior to detect run-time errors or intrusions or
- Interacting with objects whose interfaces did not exist when a distributed application was deployed initially.

With conventional DOC middleware, applying these types of changes requires tedious and error-prone re-design and re-implementation of existing application software. Moreover, it is not possible to make some of these changes efficiently if the middleware itself is not available in source code format. For example, certain middleware internals may need to be instrumented and modified to support modifications to security, buffering, feedback control, protocol, and monitoring mechanisms.

Meta-programming mechanisms help to address the limitations of conventional DOC middleware outlined above. This article describes and evaluates meta-programming mechanisms that DOC middleware can apply to help improve the adaptability and flexibility of distributed applications, while avoiding obtrusive changes to existing applications and/or DOC middleware. For concreteness, we focus on meta-programming mechanisms available with CORBA, though similar mechanisms are supported by other DOC middleware, such as COM+ and Java RMI. The meta-programming mechanisms presented in this article can be grouped into the following three categories:

1. Meta-programming mechanisms for developing and scripting generic applications. These mechanisms include:

- *The dynamic invocation interface (DII)*, which allows clients to generate requests at run-time. This mechanism is useful for interpretive applications (such as those scripted with CorbaScript [5]) that cannot have compile-time knowledge of the interfaces they access.
- *The dynamic skeleton interface (DSI):* The DSI is the server's analogue to the client's DII. The DSI allows an Object Request Broker (ORB) to deliver requests to object implementations that have no compile-time knowledge of the interfaces they implement defined using CORBA's Interface Definition Language (IDL).
- *Interface repositories*, which provide run-time information about IDL interfaces. Using this information, it is possible for an application to interact with an object whose interface was not known when the application was compiled. Interface repositories allow applications to (1) determine what operations are valid on an object, (2) make invocations on it using the DII, and (3) implement the object's interface via the DSI.

Section 3.1 describes the DII, DSI, and interface repository meta-programming mechanisms in more detail.

2. Meta-programming mechanisms that enable the context and behavior of applications to change without affecting the application implementation itself. These mechanisms include:

- *Smart proxies*, which allow distributed applications to customize their client-side behavior on a per-interface basis. Smart proxies are application-provided stub[1] implementations that transparently override the default stubs created by an ORB's IDL compiler.
- *Interceptors*, which are objects that an ORB invokes in the path of an operation invocation to monitor or modify the behavior of the invocation without changing client or server application software.
- Interceptors are often used in conjunction with other meta-objects, such as *servant managers* [3], to provide *containers* [6], which manage the resources required to customize server components transparently.

Section 3.2 describes the smart proxy and interceptor meta-programming mechanisms and Section 3.3 describes the servant manager and container meta-programming mechanisms.

3. Meta-programming mechanisms that enable the context and behavior of ORB middleware to change without affecting the ORB implementation itself. These mechanisms include:

- *Pluggable protocols*, which are objects that implement the transport mechanisms used internally in an ORB [7]. Much like the UNIX System V Streams framework [8] allows new device drivers to be added into an OS kernel transparently, pluggable protocols allow application developers to use different transport mechanisms in an ORB without having to modify its internal implementation directly. Section 3.4 describes pluggable protocol meta-programming mechanisms.
- *Bridges*, which are software components that connect different ORB domains [3] or other distributed middleware technologies, such as Microsoft's COM+ or Sun's Java RMI, and enable them to interoperate with each other transparently from an application's perspective. Different ORB domains may use different security policies or transport protocols. Bridges translate the meta-information from one ORB domain into a different format that another ORB domain or distributed middleware technology understands. Section 3.5 describes bridge meta-programming mechanisms.

The meta-programming mechanisms outlined above can be used to configure new or enhanced functionality into DOC middleware applications with little or no impact on existing software. The material presented in this article is based on our experience implementing, using, and benchmarking these meta-programming mechanisms in TAO [7], which is a CORBA-compliant, open-source ORB designed to support applications with both stringent QoS *and* flexibility requirements.

[1]Stubs provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation.

Article organization: The remainder of this article is structured as follows: Section 2 provides an overview of meta-programming; Section 3 describes alternative meta-programming mechanisms that can be applied to develop and use CORBA middleware and applications; Section 4 compares these meta-programming mechanisms; and Section 5 presents concluding remarks.

2 Overview of Meta-Programming

Meta-programming is a term given to a collection of technologies designed to improve software adaptability by decoupling application behavior from the various cross-cutting aspects [9] and resources used by applications. Applying meta-programming involves identifying and dissecting programming constructs into the following entities:

- *Base-objects*, which implement certain application-centric functionality; and
- *Meta-objects*, which abstract certain properties from base-objects (such as persistence, concurrency, scheduling, atomicity, ordering, state, replication, and change notifications) and control various aspects of their behavior at run-time.

Meta-programming techniques can be used in both programming languages and middleware, as described below.

Language-based meta-programming: In most object-oriented programming languages, objects are defined by classes, and class implementations determine the behaviors of objects. Certain languages, such as Java and Smalltalk, support meta-programming natively via *meta-class* features. A meta-object is an instance of a meta-class that encapsulates the type information and class implementation of a "base-object." A base-object implements a well-defined unit of application functionality, whereas a meta-object implements a higher-order set of behavior that can monitor and control various aspects of a base object [10].

In languages that support meta-programming natively, base-objects can interact with their meta-objects to realize object behaviors. For example, the `newInstance` method in Java's `Class` class creates a new instance of an object. A `Class` is therefore a factory that determines what type of object it creates and customizes the object's behavior accordingly.

Likewise, meta-objects can interact with each other to coordinate the behavior of base-objects. For instance, methods like `DefineClass` in Java's `ClassLoader` class allow other meta-objects to determine how and when a `Class` definition is loaded, thereby controlling base-object behavior indirectly. The interfaces that define how objects/meta-objects and meta-objects/meta-objects interact are called *meta-object protocols* (MOPs) [10], which define valid interactions between base-objects and meta-objects.

Some languages provide little or no native support for meta-programming. For example, C does not support it at all and C++ just supports it natively via its run-time type identification (RTTI) feature. C and C++ programmers can still utilize meta-programming techniques, however, by applying patterns to implement their own meta-objects manually and then using these meta-objects to control the behaviors of base-objects. For example, a meta-object implementation can apply the Reflection pattern [11] to serialize object instances and recreate them later. Likewise, meta-objects can be used to encapsulate behaviors of some classes and allow class instances to determine their behaviors dynamically.

Middleware-based meta-programming: Techniques for meta-programming can also be applied to DOC middleware, where various aspects of client/server behavior can be affected by meta-objects, such as the smart proxies, interceptors, and interface repositories described in this article. Figure 1 illustrates how client meta-objects are used in CORBA to forward

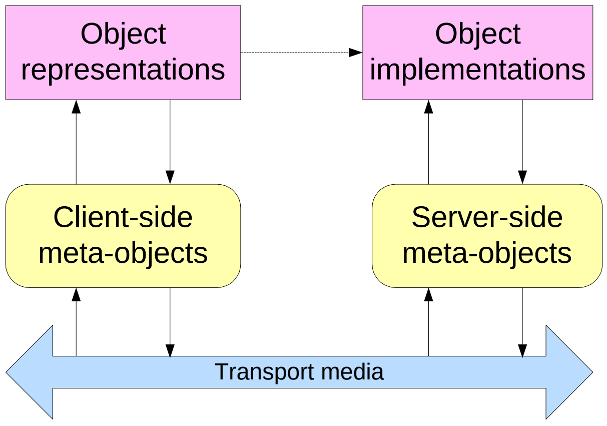


Figure 1: Meta-objects in CORBA Middleware

operation invocations to a server object implementation and how the object implementation returns the operation results via server meta-objects. From a client application perspective, its stub meta-object represents the remote object; whereas a skeleton meta-object represents the invoking client to a server application.

The meta-objects shown in Figure 1 represent a higher level of control than the base-objects that perform application-specific processing. For example, meta-objects can help connect clients to their remote server (base-)objects. They can also coordinate resources used by DOC middleware in support of client and server applications end-to-end.

Meta-programming has become prevalent in DOC middleware R&D. For example, the *Quality Object* (QuO) middleware [12] developed at BBN Technologies applies meta-programming techniques, such as smart proxies, interceptors, and bridges, to imbue regular CORBA base-objects with QoS characteristics controlled by meta-objects. Likewise, the dynamicTAO [13] reflective ORB applies meta-programming techniques to dynamically configure ORB properties for concurrency, scheduling, security, and monitoring. As these R&D activities transition into COTS middleware, such as TAO, distributed application developers can increasingly benefit from knowledge of the meta-programming mechanisms described in this article.

3 Alternative DOC Middleware Meta-Programming Mechanisms

Figure 2 provides a road map of meta-programming mechanisms we discuss in this article. As described in Section 2,

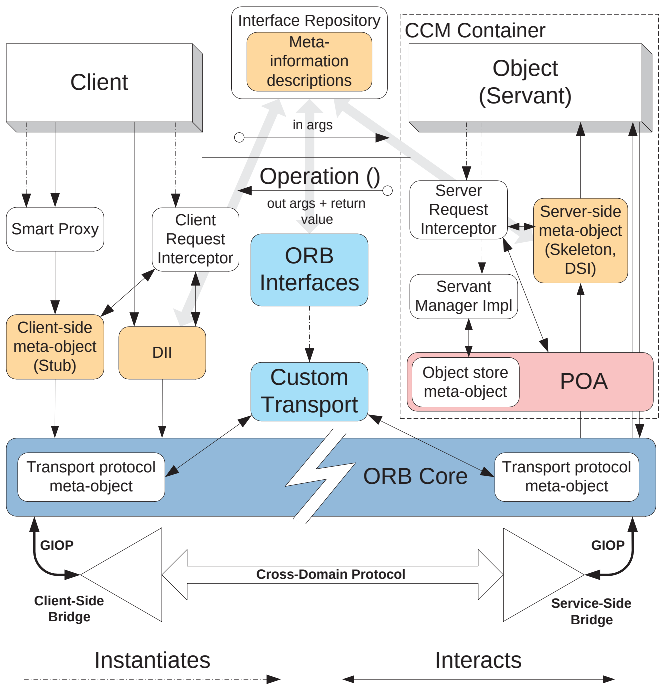


Figure 2: End-to-End Interactions Between CORBA Requests and Meta-objects

an ORB provides a meta-object framework to facilitate communication between clients and servers. The following meta-programming mechanisms can be used at different levels to enhance application performance and adaptability end-to-end:

- DII, DSI, and interface repositories provide an application with direct access to the meta-object system and are often used to implement ORB-level bridges
- Smart proxies and interceptors can alter object behavior selectively
- Pluggable protocols control how ORBs exchange messages and
- servant managers and CORBA Component Model (CCM) [6] containers coordinate resource management on the server.

The remainder of this section describes the key capabilities provided by the meta-objects illustrated in Figure 2.

3.1 DII, DSI, and Interface Repositories

Meta-objects are commonly used in DOC middleware to provide local representations for clients and remote objects in distributed object computing systems. For example, a client process interacts with a remote object via its client-side meta-object, which is called a *stub* in CORBA terminology. Likewise, a CORBA *skeleton* is a meta-object that invokes the target object implementation's operation on the client's behalf and represents the client for the duration of this upcall. These meta-objects are created by an ORB automatically as a consequence of using an interface definition language (IDL) to define component interfaces. In CORBA, for example, an IDL compiler transforms OMG IDL definitions supplied by developers into stubs and skeletons written using a particular programming language, such as C++ or Java.

In addition to providing programming language and platform transparency, an IDL compiler helps eliminate common sources of network programming errors and provides opportunities for automated compiler optimizations. CORBA applications can use these IDL compiler-generated meta-objects by linking their implementations into applications directly. While this approach is straightforward, it requires that IDL compiler-generated meta-objects be available in advance and limits applications to use a pre-determined number of interfaces.

The dynamic invocation interface (DII): For certain types of applications, such as debugging and management services, it is infeasible to know what interfaces they will encounter *a priori*. To support these applications effectively, therefore, CORBA defines a *dynamic invocation interface* (DII) mechanism that allows clients to construct invocation requests dynamically and use them to invoke the specified operations on remote objects. The DII API in CORBA is essentially a "meta-meta" mechanism that application developers can use to program the behavior of stub meta-objects dynamically. Thus, client application developers can use the DII API to invoke operations on objects whose interfaces are unknown at compile-time.

The dynamic skeleton interface (DSI): Server application developers may also need to process operation invocations on objects that a server has no built-in knowledge about at compile-time. To support this use-case, therefore, CORBA provides a *dynamic skeleton interface* (DSI). The DSI allows a server application at run-time to

- Use dynamically acquired meta-information describing the type of object it will support and
- Use this information to process invocations on this object accordingly.

Interface repositories: One purpose of DII/DSI is to defer an application's binding onto specific interface types until run-time. Ensuring the type-safety of this run-time binding requires an *interface repository*, which is a service that provides run-time information about component interfaces. For example, CORBA applications can query an interface repository to obtain meta-information that describes IDL interface types, operation signatures, operation arguments and return types, and the definition of user-defined data types. Both DII clients and DSI servers can use the meta-information provided in an interface repository to construct "generic" applications whose behavior can be determine at run-time, *e.g.* via an interpretive language like CorbaScript [5]. Moreover, these generic applications can be used to reduce the effort required to develop *bridges*, which are described in Section 3.5.

3.2 Smart Proxies and Portable Interceptors

As mentioned in Section 3.1, *stub* and *skeleton* meta-objects in CORBA serve as the "glue" between the clients and servants, respectively, and the ORB. This glue shields application developers from tedious and error-prone network programming details needed to transmit client operation invocations to server object implementations. For example, CORBA stubs implement the *Proxy* pattern [2] and marshal operation information and data type parameters into a standardized request format. Likewise, CORBA skeletons implement the *Adapter* pattern [2] and demarshal the operation information and typed parameters stored in the standardized request format.

Traditionally, the stubs and skeletons generated by an IDL compiler are *fixed*, *i.e.*, the code emitted by the IDL compiler is determined entirely at interface translation time. Fixing the generation of stubs and skeletons at this early stage, however, makes it hard for certain types of applications to adapt readily to new requirements, such as:

- The need to monitor system resource utilization may not be recognized until after an application has been deployed
- To execute securely in a particular environment, certain remote operations may require additional parameters, such as authentication certificates
- The priority at which clients invoke requests or servers handle requests may vary according to environmental conditions, such as the amount of CPU or network bandwidth available at run-time [14].

applying these types of changes to CORBA applications that use conventional fixed stubs and skeletons often requires considerable re-engineering and re-structuring of existing application software. One way to minimize the impact of these changes is to write the clients and servers entirely using the DII and DSI meta-programming mechanisms described in Section 3.1. While this approach is quite flexible, it also incurs a non-trivial time and space overhead due to the dynamic memory allocation and data copying associated with DII and DSI.

In many cases, therefore, it may be more effective to apply other meta-programming mechanisms that allow applications to adapt to various types of changes with little or no modifications to existing software. In particular, stubs and skeletons (as well as certain other points in the end-to-end operation invocation path) can be treated as *meta-objects* [10]. As operation invocations pass through these meta-objects, certain aspects of application and middleware behavior can be adapted transparently (*e.g.*, when system requirements and environmental conditions change) just by modifying the meta-objects.

To modify meta-objects, the DOC middleware can either:

- Provide mechanisms for developers to install customized meta-objects into clients or
- Embed *hooks* implementing a *meta-object protocol* (MOP) [10] into the meta-objects and provide mechanisms to install meta-objects implementing the MOP to strategize these meta-object behaviors.

In the context of CORBA, *smart proxies* are customized meta-objects and *interceptors* are meta-objects that implement the MOP. We explore both of these meta-programming mechanisms below.

3.2.1 Overview of Smart Proxies

Most CORBA application developers use the fixed stubs generated by an IDL compiler without concern for how the stubs are implemented. There are situations, however, where the default stub behavior is inadequate. For example, an application developer may wish to transparently change stub code in order to:

- Perform application-specific functionalities, such as logging
- Add parameters to a request
- Cache requests or replies to enable batch transfer or minimize calls to a remote target object, respectively or
- Support advanced QoS features, such as load balancing and fault-tolerance.

To support these capabilities *without* modifying existing client code, applications must be able to override the default stub implementations selectively. These application-defined stubs are called *smart proxies*, which are customizable meta-objects that can mediate access to target objects in a server more flexibly than the default stubs generated by an IDL compiler. Smart proxies allow developers to modify the behavior of interfaces without re-implementing client applications or target objects. Although not yet standardized by the OMG, many ORBs provide smart proxy mechanisms.

The two main entities in smart proxy designs are the *smart proxy factory* and the *smart proxy meta-object*, which are shown in Figure 3. When developers use smart proxies to

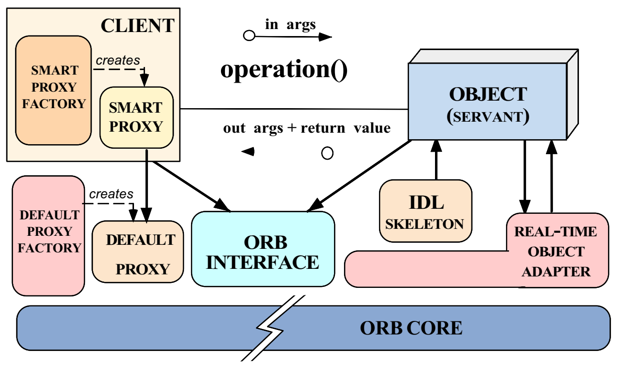


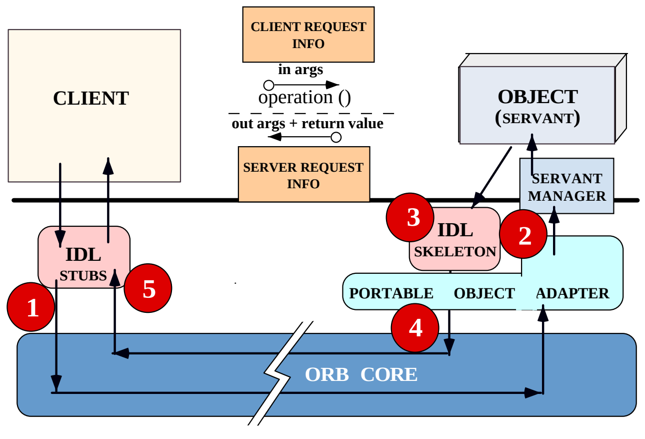
Figure 3: TAO's Smart Proxy Model

modify the behavior of interfaces, they implement smart proxy factory classes and register them with the ORB. After installing an application-supplied smart proxy factory, the ORB automatically uses the factory to create object references when a client invokes the `_narrow` operation on an interface. Thus, if smart proxies are installed before a client accesses these interfaces, the client application can transparently use the new behavior of the proxy returned by the factory.

3.2.2 Overview of Portable Interceptors

The smart proxies feature outlined above is a meta-programming mechanism that increases client application flexibility. *Interceptors* are another meta-programming mechanism used in DOC middleware to increase client *and* server application flexibility. CORBA interceptors are standard meta-objects that stubs, skeletons, and certain points in the end-to-end operation invocation path can invoke at predefined "interception points." The two types of interceptors defined in the CORBA Portable Interceptors specification [15] are described below.

1. Request interceptors: Request interceptors consist of *client request* interceptors and *server request* interceptors, which intercept the flow of a request/reply sequence through the ORB at specific points in clients and servers, respectively. Developers can install instances of these interceptors into an ORB via an IDL interface defined by the Portable Interceptor specification. Regardless of what interface an operation is invoked by, after request interceptors are installed they will be called on *every* operation invocation at the pre-determined ORB interception points shown in Figure 4.



PORTABLE INTERCEPTOR API:
```
1) send_request()/send_poll()
2) receive_request_service_contexts ()
3) receive_request()
4) send_reply()/send_exception()/send_other()
5) receive_reply()/receive_exception()/
   receive_other()
```

Figure 4: Request Interception Points in the CORBA Portable Interceptor Specification

As shown in this figure, request interception points occur in several parts of the end-to-end invocation path: when a client sends a request, when a server receives a request, when a server sends a reply, and when a client receives a reply. Different hook methods will be called at each point in this interceptor chain to allow modification of object behaviors, *e.g.*, by throwing an exception or forwarding a request to another server. The behavior of an interceptor is defined by application developers. An interceptor can examine the state of the request it is associated with and perform various actions based on the state. Interceptors can also be used to insert and extract *out-of-band* data into and out of a request invocations, respectively, via a `ServiceContextList` encapsulated with each request.

2. Interoperable Object Reference (IOR) interceptors: Version 1.1 of the CORBA Internet Inter-ORB Protocol (IIOP) introduced an IDL attribute called `components`, which contains a list of *tagged components* to be embedded within an IOR. When an IOR is created, tagged components provide a placeholder for an ORB to store additional information pertinent to the object. This information can contain various types of QoS-related information pertaining to security, server thread priorities, network connections, CORBA policies, or other domain-specific data.

IOR interceptors are objects invoked by the ORB when it creates IORs. They allow an IOR to be customized, *e.g.*, by appending tagged components. Whereas request interceptors access operation-related information via `RequestInfo` objects, IOR interceptors access IOR-related information via `IORInfo` objects. Figure 5 illustrates the behavior of IOR interceptors.

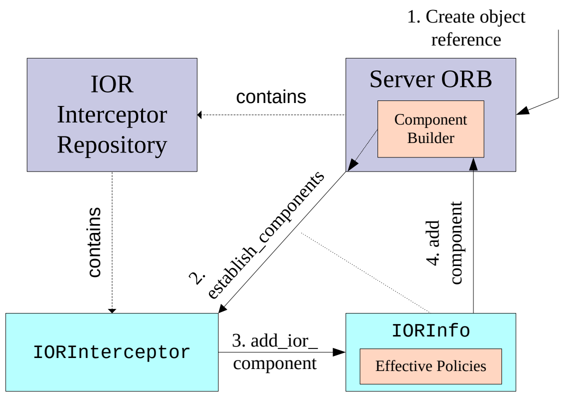


Figure 5: IOR Interceptors

3.3 Servant Managers and Containers

The CORBA specification [3] defines Portable Object Adapters (POAs) that allow server applications to register servant managers. Servant managers allow server application developers to strategize the selection, loading, unloading, and activation of object implementations. There are two types of servant managers in CORBA:

1. Servant activators: This meta-object provides two hook methods called `incarnate` and `etherealize`. The `incarnate` method indicates that a servant must be created the *first time* an object is accessed by a client. The `etherealize` method indicates when a servant can be destroyed and its resources reclaimed when the servant is deactivated and no longer needed.

2. Servant locators: This meta-object provides hook methods called `preinvoke` and `postinvoke`. The `preinvoke` method is invoked by a POA to locate a servant for *each request* on an object. The `postinvoke` method notifies the servant locator after each request that a servant is no longer in use.

Figure 6 illustrates how servant locators can be used in a CORBA application to perform various resource management activities before dispatching an operation to a servant. As shown in this figure, a POA can use a servant locator to link an implementation for an object dynamically when an operation request first arrives, thereby minimizing the resources committed to inactive servants. Other meta-mechanisms, such as the *evictor* shown in Figure 6, can help reduce system resource usage by deactivating and unlinking infrequently used servants.

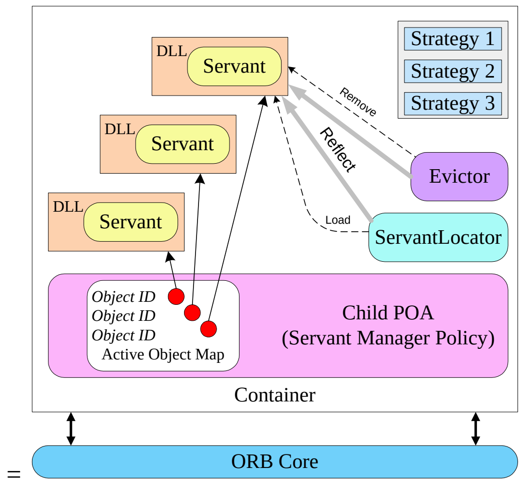


Figure 6: Managing Resources with a Servant Locator

A servant manager is similar to a server-side interceptor in several respects. For example, both implement the Interceptor pattern [1]. Moreover, both can intercept requests before they are dispatched to servants, invoke additional operations, and affect the outcome of request invocations, *e.g.*, by throwing exceptions.

Unlike interceptors, however, servant managers only affect the POAs that install them and can therefore only provide access to a limited subset of request-related information. As a result, they are more tightly coupled with POAs and servant implementations than are interceptors. For example, servant managers are used primarily to coordinate the resources necessary to activate and deactivate servants, rather than modifying internal ORB behavior.

The CORBA Component Model (CCM) [6] introduced the concept of a *container*, which is a meta-programming mechanism that provides the run-time environment for component implementations. Containers decouple application component logic from the configuration, initialization, and administration of servers. As shown in Figure 6, a CCM container creates the POA and servant locator required to activate and control a component. Standard CCM containers can be extended to implement various resource management activities, such as automatic load balancing, persistence, transactional properties, event dispatching, and QoS adaptation, without changing the behavior of application components (which play the role of base-objects).

3.4 Pluggable Protocols

Pluggable protocols frameworks [7] are another type of meta-programming mechanism provided by DOC middleware. These frameworks can be used to decouple an ORB's transport protocols from its higher-level component architecture. Developers can therefore add new protocols without changing existing middleware or application software. Pluggable protocols frameworks provide meta-objects that control the behavior of base-objects, which in this case are an ORB's message delivery mechanisms.

Figure 7 illustrates the architecture of a pluggable protocols framework that allows developers to install custom transport protocol implementations into The ACE ORB (TAO). TAO's

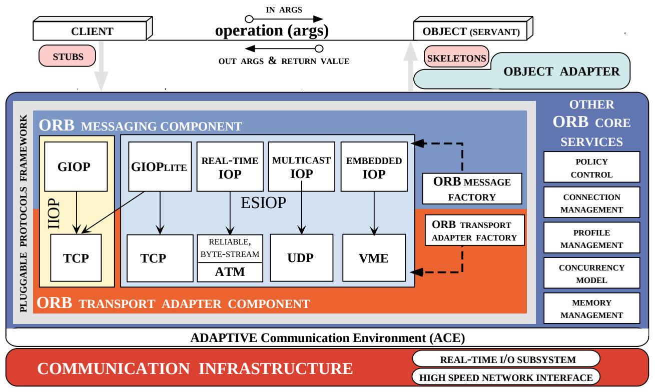


Figure 7: TAO's Pluggable Protocols Framework Architecture

pluggable protocol framework is designed to control the following two levels of protocols:

- *ORB messaging protocols*, which define the messaging format an ORB uses to exchange meta-information with other ORBs to facilitate object requests/replies, locate object implementations, and manage communication channels.
- *ORB transport protocol adapters*, which define the underlying message transport mechanism an ORB uses to exchange meta-information with other ORBs.

Separating these two protocol levels allows TAO to

- Utilize high-speed transport protocols, such as AAL5 over ATM and
- Eliminate unnecessary overhead and features in CORBA's General Inter-ORB Protocol (GIOP), which is the standard interoperability protocol that CORBA clients and servers use to exchange requests and replies.

Higher-level application components and CORBA services can use the Component Configurator pattern [1] to dynamically configure custom protocols into TAO's pluggable protocols framework without requiring obtrusive changes to themselves or the ORB.

Pluggable protocol capabilities are being standardized by the OMG in the Extensible Transport Framework [16] (ETF) specification effort. Unlike TAO's pluggable protocols framework, which allows both ORB messaging and transport protocol to be configured, the ETF specification only standardizes the interface to install the transport adaptation factory portion depicted in Figure 7. The reason for this restriction is because the ETF specification focuses primarily on Real-time CORBA ORBs that use non-TCP/IP protocols to ensure deterministic message delivery. Thus, GIOP remains the sole ORB messaging protocol standard and is not itself pluggable in the ETF specification.

3.5 Bridges

An ORB domain is a group of inter-connected ORBs that have the same administrative policies, such as security policies, use the same ORB messaging and transport protocols, and are capable of invoking operations directly upon one another. In contrast, ORBs that reside in different domains cannot invoke operations on each other directly because they may not be using the same protocols or direct invocations may violate certain administrative policies, *e.g.*, such as Internet firewall restrictions.

To allow inter-domain communication, the CORBA specification defines the concept of *bridges*. A bridge can connect an ORB in one ORB domain to other ORBs in different ORB domain or other systems running different distributed middleware technologies, such as COM+ [17] or Java RMI. It can also connect different ORB domains and allow invocations across domain boundaries by translating these requests.

Figure 8 illustrates how bridges can be used to allow

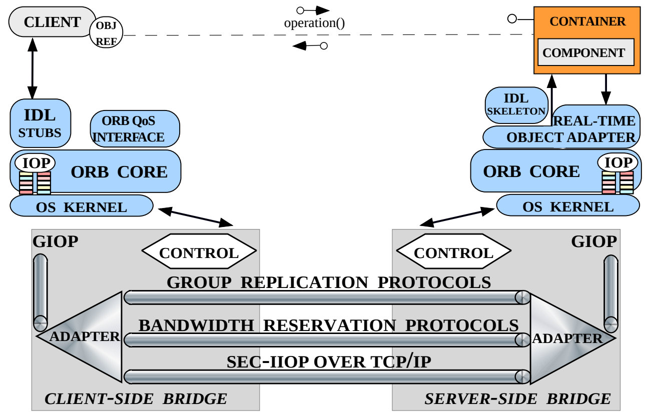


Figure 8: Applying Bridges to Inter-connect ORBs Across Separate Domains

CORBA applications to communicate across a wide range of communication links, ranging from firewalled TCP/IP to wide-area group communication protocols. In the context of this article, a bridge can be viewed as a high-level meta-object that handles requests by using their meta-data to translate them into different protocols, thereby allowing its client and server base-objects to interact end-to-end.

It is not feasible to implement a *statically configured* generic bridge that can receive, process, and forward all types of requests. Many CORBA bridge implementations therefore use the DII, DSI, and interface repositories described in Section 3.1. For example, a generic bridge can use the DII and DSI to process a two-way operation as follows:

1. Receive a request and use the DSI API and an interface repository to decompose the request into a name/value list (`NVList`), which is a standard CORBA construct that stores each argument in a separate `any` data structure
2. Use the same `NVList` to construct a DII request and use the DII API to send this request to the target object
3. When reply is received the bridge uses the DII request object to extract the return value and output arguments
4. The DII object is then used to fill in the return value and output arguments for the original DSI request.

It should be clear from the discussion above that an ORB bridge architecture can incur significant overhead. In particular, an ORB may require multiple data copies, *e.g.,* first copying the data from the request buffer into each one of the `any`s of the DSI argument list and then later copying the reply from the DSI argument list into the DII request buffer. Copies are also required from the DII reply buffer into the `any`s used to extract each argument and finally into the DSI reply buffer.

4 Comparing ORB Middleware Meta-Programming Mechanisms

Section 3 describes a range of middleware-oriented meta-programming mechanisms—*i.e.*, DII/DSI, interface repositories, smart proxies, interceptors, servant managers, pluggable protocols, and bridges—that we have implemented in TAO. These meta-programming mechanisms allow CORBA applications to adapt to requirement or environmental changes that occur late in an application's life-cycle with little or no obtrusive changes to existing software. Based on our experience implementing and using the meta-programming mechanisms in TAO, we have observed the tradeoffs and limitations described in this section.

4.1 Generality vs. Overhead

To select suitable meta-programming mechanisms, developers must understand the tradeoffs between generality vs. overhead in their applications. Typically, the greater the generality of a meta-programming mechanism, the greater its overhead with respect to non-functional properties, such as static/dynamic memory footprint, function-call indirection, or CPU consumption. Thus, DSI/DII, interface repositories, and bridges often incur the most overhead since they are the most general meta-programming mechanisms.

The other mechanisms presented in this article incur less overhead, ranked roughly in terms of how closely they target specific ORB or application mechanisms. In general, the broader the scope a mechanism targets, the greater the overhead incurred by the mechanism. For example, portable interceptors often incur more overhead than smart proxies because they influence operation processing at multiple points along an invocation path. Thus, they must handle all interfaces via the CORBA generic `any` type, whereas smart proxies only affect the specific interfaces they target.

Although reducing overhead is often worthwhile, the flexibility afforded by the more general meta-programming mechanisms may be important for certain types of applications. For example, it may be infeasible for network management applications to know at compile-time all the types of object schemas in a management information base (MIB). To solve this problem generically therefore necessitates some type of DII and interface repository to navigate arbitrary MIBs at run-time.

Often, the additional indirection incurred by certain meta-programming mechanisms may be a small cost relative to the large potential gain in performance at a higher level. For example, although pluggable protocols frameworks require additional levels of indirection, *e.g.,* due to virtual method calls, the ability to install and use new protocols rapidly can improve end-to-end performance by several orders of magnitude [7]. Moreover, a well-crafted ORB implementation can minimize overhead when certain features are not used by the application at run-time. When combined with patterns (such as Component Configurator [1] or Reflection [11]) and OS features (such as explicit dynamic linking), meta-programming mechanisms can be configured selectively and dynamically into CORBA clients and servers.

4.2 Early vs. Late Lifecycle Integration

Different meta-programming mechanisms can be introduced at different stages in an application's lifecycle. For example, bridges and pluggable protocols can be (re)configured transparently into applications. Moreover, some ORBs allow new protocols to be linked into the ORB at run-time [7]. Smart proxies, portable interceptors, and servant managers can be retrofitted into applications with minimal impact on existing code.

DII and DSI share some commonalities with portable interceptors, *e.g.,* they can all be used on any interface and cannot modify argument values directly. Behavioral adapation can therefore be achieved by changing the meta-information they received at run-time. However, unlike portable interceptors—which can be installed without changing application implementations—the use of DII/DSI must be selected during an application's design phase. This constraint can be offset to some extent by using DII/DSI in conjunction with scripting languages, such as CorbaScript [5], although the use of scripting languages can degrade performance relative to compiled applications.

4.3 Portability

If an application must run on multiple ORB implementations, portability is an important criteria when selecting meta-programming mechanisms. The meta-programming mechanisms described in this article can be grouped into the three general portability categories shown in Table 1

Standardization Level	**Features**
Standardized	DII/DSI, interface repositories, servant managers, and bridges have been defined in the CORBA specification for many years. They should therefore work with any CORBA-conformant ORB. The portable interceptor [15] and CCM container [6] specifications have being adopted recently and many ORBs should support these features soon.
Standardization underway	The extensible transport framework [16] is in the process of being standardized by the OMG. At the present time, however, many ORBs provide proprietary implementations of this feature.
Non-standard	Smart proxies are not currently part of the CORBA standard. However, many ORBs provide some form of smart proxy mechanism as an extension.

Table 1: Meta-programming Mechanism Portability

5 Concluding Remarks

DOC middleware has been applied successfully to many domains. Its primary benefits are:

- Shielding developers from distributed computing and communication challenges, such as security, partial failure, and end-to-end QoS-enabled resource management; and
- Allowing applications to invoke operations on target objects efficiently without concern for their location, programming language, OS platform, communication protocols and interconnects, and hardware [4].

Historically, however, many DOC middleware solutions have tightly coupled interfaces and implementations. This coupling makes it hard to adapt middleware and applications to changes in requirements and environmental conditions that occur late in a system's life-cycle, *i.e.*, after deployment and/or at run-time.

The meta-programming mechanisms described in this article are becoming commonplace in DOC middleware. In CORBA, for example, ORBs are responsible for transmitting client operation invocations to target objects and returning their replies (if any). When a client invokes an operation, it interacts with a stub meta-object directly or through a smart proxy. The stub works in conjunction with transport-protocol meta-objects controlled by a pluggable protocol framework to access and/or transform (as in the case of a CORBA bridge), a client operation invocation into a request message and transmit it to a server. On the server's request processing path various meta-objects, such as interceptors and servant managers, can then access and/or perform inverse transformations on the operation invocation message and dispatch the message to its servant. An invocation result is returned in a similar fashion in the reverse direction.

The growing availability and use of meta-programming mechanisms is helping to increase the flexibility and adaptability of DOC middleware and applications. The meta-programming mechanisms described in this article are all implemented in the TAO open-source CORBA ORB, which can be downloaded from `www.cs.wustl.edu/$\sim$schmidt/TAO.html`. Our experience using TAO on many commercial and research projects indicates that improvements in flexibility and adaptability can occur without significant performance degradation if developers select their meta-programming mechanisms carefully.

Future challenges confronting researchers and developers involve identifying the appropriate patterns, protocols, and architectures that can be applied to devise *policies* for configuring and controlling meta-programming mechanisms robustly and efficiently. Another important theme underlying future meta-programming efforts is *adaptive and reflective middleware* [14]. Adaptive middleware is software whose functional and/or QoS-related properties can be modified either:

- *Statically*, *e.g.*, to reduce footprint, leverage capabilities that exist in specific platforms, enable functional subsetting, and minimize hardware/software infrastructure dependencies; or
- *Dynamically*, *e.g.*, in response to changes in environmental conditions or requirements, such as changing component interconnection topologies; component failure or degradation; changing power-levels; changing CPU demands; changing network bandwidth and latencies; and changing priority, security, and dependability needs.

Reflective middleware goes a step further to permit automated examination of the capabilities it offers, and then permits automated adjustment to optimize those capabilities [12, 13]. Reflective middleware supports a more advanced form of adaptive behavior, in that the necessary adaptations can be performed autonomously (or semi-autonomously) based on dynamic conditions within the system, in the system's environment, or in the policies defined by system users. Such automatic adaptations must clearly be performed carefully to ensure that distributed optimizations retain distributed system stability and converge rapidly.

Acknowledgements

We thank Jeff McNiel, Philippe Merle, Jerry Odenwelder, and the anonymous reviewers for comments that helped improve this article. In addition, we thank John Zinky for the initial discussions that motivated us to write this article.

References

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

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

[3] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.

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

[5] P. Merle, C. Gransart, and J.-M. Geib, "Generic Tools: A New Way to Use CORBA," in *Workshop on CORBA: Implementation, Use and Evaluation at ECOOP '97*, (Jyväskyä, Finland), June 1997.

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

[7] C. O'Ryan, F. Kuhns, D. C. Schmidt, and J. Parsons, "Applying Patterns to Develop a Pluggable Protocols Framework for ORB Middleware," in *Design Patterns in Communications* (L. Rising, ed.), Cambridge University Press, 2000.

[8] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

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

[10] C. Zimmermann, "Metalevels, MOPs and What the Fuzz is All About," in *Advances in Object-Oriented Metalevel Architectures and Reflection* (C. Zimmermann, ed.), pp. 3–24, Boca Raton, FL: CRC Press, 1996.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley and Sons, 1996.

[12] 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, pp. 1–20, 1997.

[13] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[14] N. Wang, D. C. Schmidt, M. Kircher, and K. Parameswaran, "Adaptive and Reflective Middleware for QoS-Enabled CCM Applications," *IEEE Distributed Systems Online*, vol. 2, July 2001.

[15] Object Management Group, *Interceptors FTF Final Published Draft*, OMG Document ptc/00-04-05 ed., April 2000.

[16] Object Management Group, *Extensible Transport Framework for Real-Time CORBA, Request for Proposal*. Object Management Group, OMG Document orbos/2000-09-12 ed., Feb. 2000.

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