The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware

Nanbor Wang Kirthika Parameswaran Douglas Schmidt Ossama Othman

{nanbor, kirthika}@cs.wustl.edu
Department of Computer Science
Washington University, St.Louis

{schmidt, ossama}@uci.edu
Electrical & Computer Engineering
University of California, Irvine

Abstract

Distributed object computing (DOC) middleware shields developers from many tedious and error-prone aspects of programming distributed applications. Without proper support from the middleware, however, it can be hard to evolve distributed applications after they are deployed. Therefore, DOC middleware should support meta-programming mechanisms, such as smart proxies and interceptors, that improve the adaptability of distributed applications by allowing their behavior to be modified without changing existing software drastically.

This paper presents three contributions to the study of meta-programming mechanisms for DOC middleware. First, it illustrates, compares, and contrasts several meta-programming mechanisms from an application developer's perspective. Second, it outlines the key design and implementation challenges associated with developing smart proxies and portable interceptors features for CORBA. Third, it presents empirical results that pinpoint the performance impact of smart proxies and interceptors. Our goal is to help researchers and developers determine which meta-programming mechanisms best suit their application requirements.

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 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 interfaces, that provide location transparency to clients and server components;

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

3. Amortizing software lifecycle costs by leveraging previous development expertise and capturing implementations of key patterns in reusable middleware frameworks and common services.

In the case of standards-based DOC middleware, such as CORBA [1], these capabilities are realized via an open specification process. The resulting products can interoperate across many OS/network platforms and programming languages [2].

To date, CORBA middleware has been used successfully to enable developers to create applications rapidly that can meet a particular set of requirements with a reasonable amount of effort. CORBA has been less successful, however, at shielding developers from the effects of requirement or environmental changes that occur late in an application's life-cycle, *i.e.*, during deployment and/or at run-time. To address this problem, this paper describes and evaluates *meta-programming mechanisms*, which improve the adaptability of distributed applications by allowing their behavior to be modified with little or not change to existing application software.

The two meta-programming mechanisms we focus on in this paper are:

• **Smart proxies,** which are application-provided stub implementations that transparently override the default stubs created by an ORB to customize client behavior on a per-interface basis.

• **Interceptors,** which are objects that an ORB invokes in the path of an operation invocation to monitor or modify the behavior of the invocation transparently.

These two meta-programming mechanisms can be used to configure new or enhanced functionality into CORBA applications with minimal impact on existing software. The material presented in this paper is based on our experience implementing and using smart proxies and interceptors in TAO [3], which is a open-source, CORBA-complaint ORB designed to support applications with demanding quality-of-service (QoS) requirements.

Paper organization: The remainder of this paper is structured as follows: Section 2 presents an overview of the *smart proxy* and *interceptor* meta-programming mechanisms; Section 3 illustrates how to use smart proxies and interceptors; Section 4 describe the patterns that guided the development of TAO's smart proxy and interceptor mechanisms and resolved key design challenges; Section 5 illustrates the performance characteristics of TAO's smart proxy and interceptor mechanisms; Section 6 compares our work with related research; and Section 7 presents concluding remarks.

2 Overview of Smart Proxies and Interceptors

DOC middleware provides *stub* and *skeleton* mechanisms that serve as a "glue" between the client and servants, respectively, and the ORB. For example, CORBA stubs implement the *Proxy* pattern [4] and marshal operation information and data type parameters into a standardized request format. Likewise, CORBA skeletons implement the *Adapter* pattern [4] and demarshal the operation information and typed parameters stored in the standardized request format.

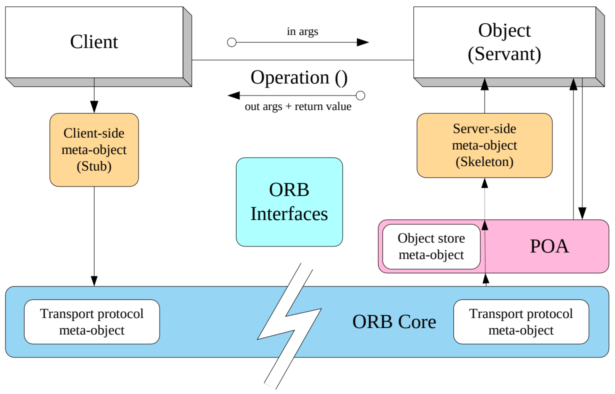
CORBA stubs and skeletons can be generated automatically from schemas defined using the OMG Interface Definition Language (IDL). A CORBA IDL compiler transforms application-supplied OMG IDL definitions into stubs and skeletons written using a particular programming language, such as C++ or Java. In addition to providing programming language and platform transparency, an IDL compiler eliminates common sources of network programming errors and provides opportunities for automated compiler optimizations [5].

Traditionally, the stubs and skeletons generated by an IDL compiler are *fixed*, *i.e.*, the code emitted by the IDL compiler is determined at translation time. This design shields application developers from the tedious and error-prone network programming details needed to transmit client operation invocations to server object implementations. Fixed stubs and skeletons make it hard, however, for applications to adapt readily to certain types of changes in requirement or environmental conditions, such as:

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

In applications based on CORBA middleware with conventional fixed stubs/skeletons, these types of changes often require re-engineering and re-structuring of existing application software. One way to minimize the impact of these changes is to devise *meta-programming mechanisms* that allow applications to adapt to various types of changes with little or no modifications to existing software. For example, stubs, skeletons, and certain points in the end-to-end operation invocation path can be treated as *meta-objects* [6], which are objects that refine the capability of base-level objects, which are the objects comprising the bulk of application programs.

As shown in Figure 1, CORBA ORBs are responsible for transmitting client operation invocations to target objects. When a client invokes an operation, a stub implemented as



Client
in args
Object (Servant)
Operation ()
out args + return value
Client-side meta-object (Stub)
Server-side meta-object (Skeleton)
ORB Interfaces
Object store meta-object
POA
Transport protocol meta-object
ORB Core
Transport protocol meta-object

Figure 1: Interactions Between Requests and Meta-objects End-to-End

a meta-object can act in conjunction with transport-protocol meta-objects to access and/or transform a client operation invocation into a message and transmit it to a server. Corresponding meta-objects on the server's request processing path can access and/or perform inverse transformations on the operation invocation message and dispatch the message to its servant. An invocation result is delivered in a similar fashion in the reverse direction.

As all operation invocations pass through meta-objects, certain aspects of application and middleware behavior can be adapted transparently when system requirements and environmental conditions change by simply modifying the meta-objects. To modify meta-objects, the DOC middleware can either (1) provide mechanisms for developers to installed customized meta-objects for the client or (2) embed *hooks* implementing a *meta-object protocol* (MOP)[6] in the meta-objects and provide mechanisms to install objects implementing the MOP to strategize these meta-object behaviors. In the context of CORBA, *smart proxies* are customized meta-objects and *interceptors* are objects that implement the MOP.

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 change stub code transparently in order to:

- Perform application-specific functionality, 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;
- Support advanced quality-of-service (QoS) features, such as load balancing and fault-tolerance; or
- Enforce security mechanisms, such as authentication of credentials.

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 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.

The two main entities in smart proxy designs are (1) the smart proxy factory and (2) the smart proxy meta-object, which are shown in Figure 2. When using a smart proxy to modify the behavior of an interface, the developer implements the smart proxy class and registers it with the ORB. After installing the smart proxy factory, the ORB automatically uses the application-supplied factory to create object references when a client invokes the _narrow operation of 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.

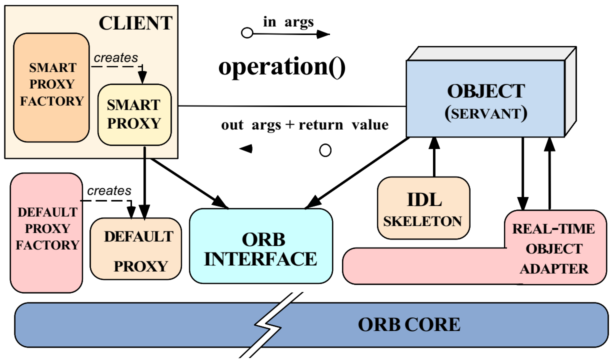


Figure 2: TAO's Smart Proxy Model

Smart proxies are not yet standardized in CORBA, though many ORBs support this feature as a proprietary extension.

2.2 Overview of Interceptors

The smart proxies feature outlined above is a meta-programming mechanism that increases the flexibility of *client* applications. *Interceptors* are another meta-programming mechanism used in DOC middleware to increase the flexibility of both client *and* server applications. In 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."

Prior to CORBA 2.3.1 interceptors were under-specified and therefore non-portable. In contrast, the interceptors discussed in this paper are based on the so-called "Portable Interceptors" specification [7], which is being ratified by the OMG. Two types of interceptors are defined in the CORBA Portable Interceptor specification:

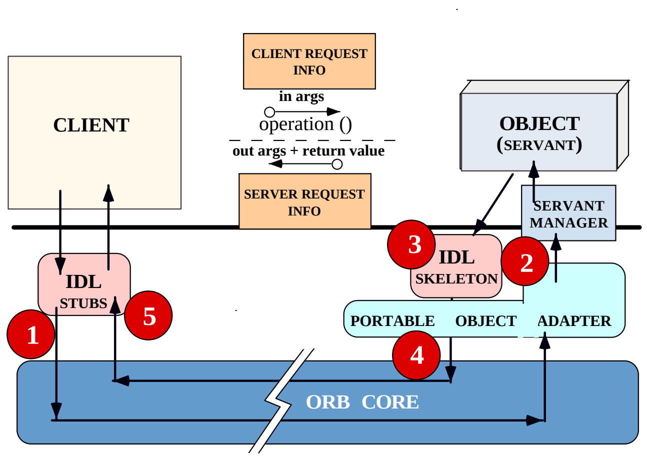
- *Request interceptors*, which deal with operation invocations;
- *IOR interceptors*, which insert information into interoperable object references (IORs).

Both types of interceptor are described below.

2.2.1 Request Interceptors

Request interceptors can be decomposed into *client request* interceptors and *server request* interceptors, which are designed to intercept the flow of a request/reply sequence through the ORB at specific points on 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 or operation is invoked, after request interceptors are installed they will be called on *every* operation invocation at the pre-determined ORB interception points shown in Figure 3.

As shown in this figure, request interception points occur in multiple 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 different points in this interceptor chain. For example, the `send_request` hook is called on the client before the request is marshaled and the `receive_request` hook is called on the server after the request is demarshaled.



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

Figure 3: Request Interception Points in the CORBA Portable Interceptor Specification

Compared to a client invocation path, a server invocation path has an additional interception point called `receive_request_service_contexts`, which is invoked before the POA dispatches a servant manager. This interception point prevents unnecessary upcalls to a servant. For example, in the CORBA Security Service [8] framework this interception point can be used to inspect security-related credentials piggybacked in a service context list entry. If the credentials are valid the upcall can proceed to other interceptors (if they exist) or to the servant; if not, an exception will be returned to the client.

The behavior of an interceptor can be defined by an application developer. An interceptor can examine the state of the request that it is associated with and perform various actions based on the state. For example, interceptors can invoke other CORBA operations, access information in a request, insert/extract piggybacked messages in a request's service context list, redirect requests to other target objects, and/or throw exceptions based on the object the original request is invoked upon and the type of the operation. Each of these capabilities is described below:

Nested invocations: A request interceptor can invoke operations on other CORBA objects before the current invocation it is intercepting completes. For example, monitoring and debugging utilities can use this feature to log information associated with each operation invocation. To avoid causing infinite recursion, developers must be careful to act only on targeting interfaces and operations they intend to affect when performing nested invocations in an interceptor.

Accessing request information: Request interceptors can access various information associated with an invocation, such as the operation name, parameters, exception lists, return values, and the request id via the MOP interface as defined in the Portable Interceptor specification. Interceptors cannot, however, modify parameters or return values. This request/reply information is encapsulated in an instance of `ClientRequestInfo` or `ServerRequestInfo` classes, which derive from the `RequestInfo` class and contain the information listed above for each invocation.

For example, client request interceptors are passed `ClientRequestInfo` and server request interceptors are passed `ServerRequestInfo`. These `RequestInfo`-derived objects can use features provided by the CORBA `Dynamic` module. This module is a combination of pseudo-IDL types, such as `RequestContext` and `Parameter`, declared in earlier CORBA specifications. These types facilitate on-demand access of request information from the `RequestInfo` to avoid unnecessary overhead if an interceptor does not need all the information available with the `RequestInfo`.

Service context manipulation: As mentioned earlier, request interceptors cannot change parameters or the return value of an operation. They can, however, manipulate *service contexts* that are piggybacked in operation requests and replies exchanged between the clients and servers. A service context is a sequence field in a GIOP message that can transmit "out-of-band" information, such as authentication credentials, transaction contexts, operation priorities, or policies associated with requests.

For example, the CORBA Security Service uses request interceptors to insert user identity via service contexts. Likewise, the CORBA Transaction Service uses request interceptors to insert transaction-related information into service contexts so it can perform extra operations, such as commit/rollback, based on the operation results in a transaction. Each service context entry has a unique service context identifier that applications and CORBA components can use to extract the appropriate service context.

Location forwarding: Request interceptors can be used to forward a request to a different location, which may or may not be known to the ORB *a priori*. This can be done via the `PortableInterceptor::ForwardRequest` exception, which allows an interceptor to inform the ORB that a retry should occur upon the new object indicated in the exception. The exception can also indicate whether the new object should be used for all future invocations or just for the forwarded request.

Since the `ForwardRequest` exception can be raised at most interception points, it can be used to provide fault tolerance and load balancing [9]. For example, the IOR of a replicated object can be used as the forward object in this exception. When the object dies for some reason–and this situation is conveyed to the interceptor–this exception can be raised even before the POA tries to make an upcall.

Multiple interceptors: Multiple request interceptors can be registered with an ORB, which will then iterate through them and invoke the appropriate interception operation at every interception point according to the following rules:

- For each request interceptor, only one *starting* interception point can be called for a given invocation. A starting interception point is the first point invoked in a request/reply sequence. For instance, the starting points for a client ORB include `send_request` and `send_poll`. Likewise, the starting point for a server ORB is `receive_request_service_contexts`.
- For each request interceptor, only one *ending* interception point can be called for a given invocation. The ending interception point is the last juncture where an interception may occur in the request/reply sequence. The ending interception points on a client ORB are `receive_reply`, `receive_exception`, and `receive_other` and the ending interception points for a server ORB consist of `send_reply`, `send_exception`, and `send_other`.
- There can be multiple intermediate interception points.
- Intermediate interception points cannot be invoked in the case of an exception.
- The ending interception point for a given interceptor will be called only if the starting interception point runs to completion.

Multiple interceptors are invoked using a flow-stack model. When initiating an operation invocation, an interceptor is pushed onto the stack after its starting interception point completes successfully. When an invocation completes, the interceptors are popped off the stack and invoked in reverse order. The flow-stack model ensures that only interceptors executed successfully for an operation can process the reply/exceptions.

Exception handling: Request interceptors can affect the outcome of a request by raising exceptions in the inbound or outbound invocation path. In such cases, the `send_exception` operation of a server request interceptor is invoked on the reply path and is received at the client in the `receive_exception` interceptor hook. When a `send_exception` or `receive_exception` operation raises a `ForwardRequest` exception, the other interceptors have their `send_other` and `receive_other` interception points invoked, respectively.

2.2.2 IOR Interceptors

IIOP version 1.1 introduced an attribute called `components`, which contains a list of *tagged component*s to be embedded within an IOR. When an IOR is created, tagged components provide a placeholder for an ORB to store extra information pertinent to the object. This information can contain various types of QoS information related to security, server thread priorities, network connections, CORBA policies, or other domain-specific information.

The original IIOP 1.0 specification provided no standard way for applications or services to add new tagged components into an IOR. Services that require this field were therefore forced to use proprietary ORB interfaces, which impeded their portability. The Portable Interceptors specification resolves this problem by defining *IOR interceptors*.

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`s, IOR interceptors access IOR-related information via `IORInfo`s. Figure 4 illustrates the behavior of IOR interceptors. A server

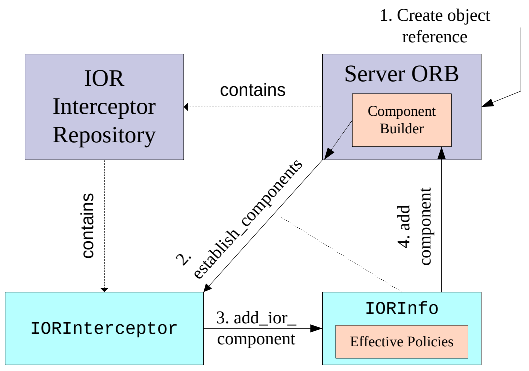


Figure 4: IOR Interceptors

ORB responsible for creating an IOR contains an *IOR interceptor repository*. In turn, this repository contains a series of IOR interceptors that have been registered with the ORB. When the server process requests the ORB to create an IOR, the ORB iterates through the IOR interceptors in the repository using the `establish_components` operation. The IOR interceptors then add tagged components to the IOR being generated by refering to the `IORInfo` passed in by calling `add_ior_component` or `add_ior_component_to_profile`.

2.3 Evaluating Alternative Meta-Programming Mechanisms for ORB Middleware

We presented an overview of smart proxies and interceptors above. We now evaluate these two mechanisms, and then compare and contrast them with two other meta-programming mechanisms–pluggable protocols and servant managers–that are provided by most CORBA implementations.

2.3.1 Smart Proxies vs. Interceptors

Smart proxies and interceptors are similar in that they extend ORB-mediated invocations and functions. They differ, however, in their architecture and have their own pros and cons, as described below.

Intent: A smart proxy can be used for a variety of purposes, such as improving performance via caching, whereas interceptors are used primarily to (1) audit and verify information along the invocation path and (2) redirect the operation if necessary. For instance, a server request interceptor can determine whether the server should handle certain operation invocations by inspecting the incoming requests and forwarding some requests to other servers that can handle them.

Scope of control: A different smart proxy can be configured for each interface, whereas the same set of interceptors will be invoked at *all* the ORB mediated points of an invocation. Moreover, a smart proxy is solely a client mechanism, whereas request interceptors are invoked on the request path from client-to-server and on the reply path from server-to-client.[1]

Invocation points: A smart proxy invocation point occurs whenever an operation is invoked through a stub. In contrast, interceptors are invoked at many points, including IOR creation time and/or before a call is sent by the POA to the servant.

Cardinality: A client can have only a single smart proxy for each interface, whereas multiple interceptors can be registered with the ORB.

Modifiability: Since smart proxies replace default ORB generated stubs completely, smart proxies can modify the parameters or results of an operation. In contrast, the Portable Interceptor specification does not allow request interceptors to change operation parameters or return values.

Overhead: A smart proxy mechanism incurs minimal overhead, *i.e.*, a single extra method call per-operation invocation. In contrast, request interceptors can incur additional overhead to access request information because information related to the request is bundled into `any`s, which have higher overhead for their insertion and extraction operations.

Standardization: Smart proxies have not yet been standardized in the CORBA specification. CORBA interceptors will be standardized after the Portable Interceptor specification is ratified.

[1] IOR interceptors are just invoked during object reference creation.

In general, design problems that require pre-invocation or per-interface extensions are well-suited for smart proxies. Portable interceptors, in contrast, provide a suitable solution for applications that require a semantically richer–albeit somewhat more expensive–meta-programming abstraction.

2.3.2 Servant Managers

The CORBA POA specification [1] allows server applications to register *servant manager* objects that activate servants on-demand, rather than creating all servants before listening for requests. There are two types of servant managers in CORBA:

- **Servant activators,** which provide a hook method called `incarnate` that creates a servant the first time an object is accessed by a client.
- **Servant locators,** which provide a hook method called `preinvoke` that are invoked by a POA to create a servant for every request on an object. Figure 5 illustrates how servant locators are used in a CORBA application to perform various resource management activities before dispatching an operation to a servant.

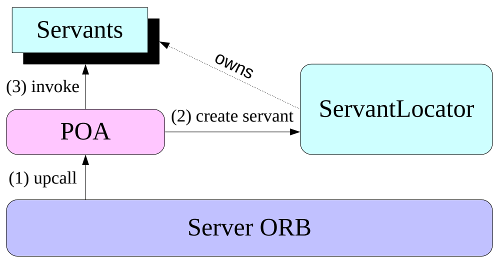


Figure 5: Managing Resources with a Servant Locator

A servant locator is similar to an interceptor in several respects. For example, both are implementations of the Interceptor pattern [10]. Moreover, both can (1) intercept requests before they are dispatched to servants, (2) invoke extra operations, and (3) affect the outcome of request invocations, *e.g.*, by throwing exceptions. Unlike interceptors, however, servant locators only affect the POAs that install them and can only provide access to a limited subset of the request-related information. As a consequence, they are more tightly coupled with POAs and servant implementations than are interceptors.

2.3.3 Pluggable Protocols Frameworks

Another type of meta-programming mechanisms provided by some DOC middleware is *pluggable protocols frameworks* [11, 12], which is in the process of being standardized by the OMG in the Extensible Transport Framework [13] specification effort. These frameworks decouple the ORB's transport protocols from its component architecture. Developers can therefore add new protocols without requiring changes to existing application software.

Figure 6 illustrates TAO's pluggable protocols framework, which allows developers to install new protocols into the ORB by implementing customized pluggable protocol objects. Higher-level application components and CORBA services

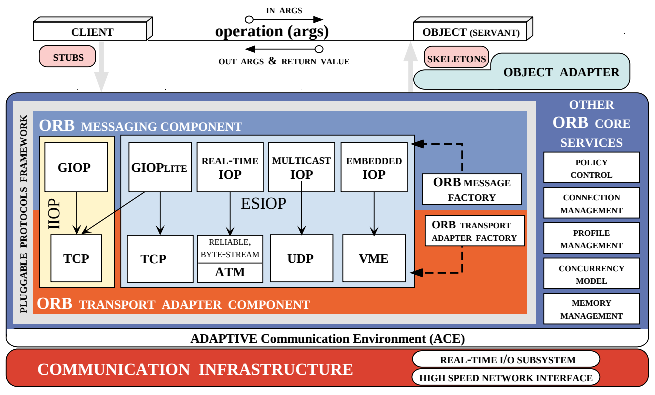


Figure 6: TAO's Pluggable Protocols Framework Architecture

use the Component Configurator pattern [10] to dynamically configure custom protocols into TAO's pluggable protocols framework *without* requiring obtrusive changes to themselves or the ORB.

As with interceptors and smart proxies, pluggable protocols frameworks are meta-programming mechanisms that add functionality to ORBs. However, whereas other two mechanisms alter the semantic of objects, pluggable protocols frameworks alter the underlying ORB transport mechanism. Thus, they do not permit fine-grained control over objects since they affect *all* objects in an ORB and it is hard to vary the transport mechanism at the level of object references. Moreover, since pluggable protocols deal directly with the communication infrastructure, they are usually more complex to program than interceptors or smart proxies.

Figure 7 compares the various meta-programming mechanisms along a number of dimensions described above. Portable interceptors have the highest overhead since they are the most flexible meta-programming mechanism. Although other mechanisms have less overhead compared to portable interceptors, they are targeted at more specific system mechanisms. When combined with patterns, such as Component Configurator [10] and OS features, such as explicit dynamic linking [14], these meta-programming mechanisms can all be configured dynamically into CORBA clients and servers.

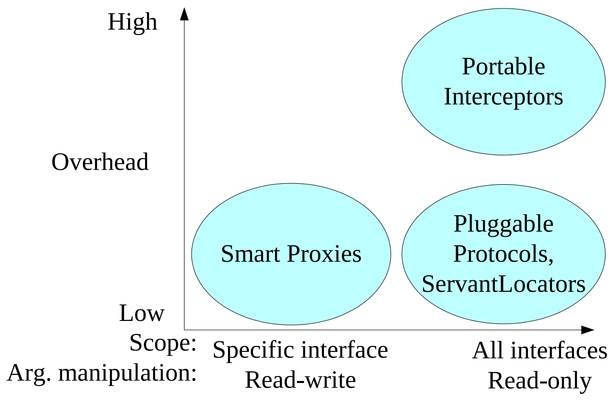


Figure 7: Comparing Alternative Meta-programming Mechanisms

3 Programming with Smart Proxies and Portable Interceptors

This section describes two examples that illustrate how the smart proxy and interceptor meta-programming mechanisms can be used to adapt existing systems as requirements change *without* impacting client and server application software appreciably. In particular, smart proxies and interceptors allow applications to modify their behavior by changing the behavior of their meta-objects, rather than by redesigning interfaces and application implementations.

3.1 Using Smart Proxies for Secure Transactions

Overview: Below, we illustrate the use of smart proxies to simplify the addition of security to a stock quote system after it has been deployed. A sample system configuration consisting of a remote database server and two clients is shown in Figure 8.

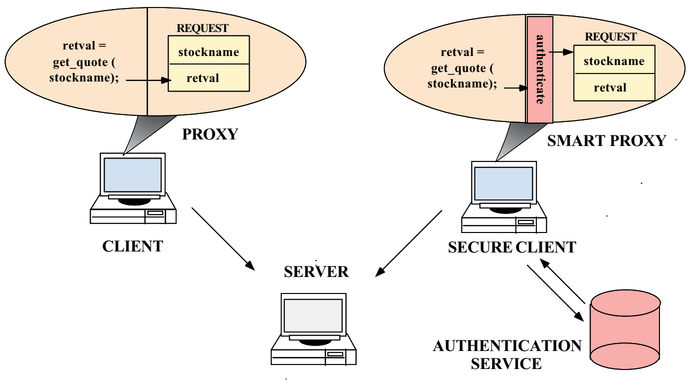


Figure 8: A Secure Transaction System Using Smart Proxies

Originally, clients accessed stock quotes via the following IDL definition:

```
module Stock
{
  // The interface for which with a smart proxy
  // will be provided for authentication of the
  // client to the Quoter for validation purposes.
  interface Quoter {
    // Exception raised when stock_name does
    // not exist.
    exception Invalid_Stock {
      string reason;
    };

    // Two-way operation to retrieve current stock
    // value.  This method will be customized by
    // the smart proxy to include authentication.
    long get_quote (in string stock_name)
      raises (Invalid_Stock);

    // One-way operation for auditing purposes.
    oneway log_quote (long quote);
  };

  // ...
};
```

Our goal is to avoid changing this existing IDL, while adding the ability to authenticate clients. By using smart proxies, we can extend the original application transparently to invoke a security mechanism that performs authentication. Moreover, if the security mechanism must be revised in the future, a new smart proxy can be used and the old one removed without affecting application code.

Generating smart proxies in TAO: TAO's IDL compiler parses IDL files containing CORBA interfaces and data types and generates stubs/proxies and skeletons, which are then integrated into client and server application code, respectively. The front-end of TAO's IDL compiler parses OMG IDL input files and generates an abstract syntax tree (AST). The back-end of TAO's IDL compiler *visits* the AST to generate CORBA-compliant C++ source code [15].

To add smart proxy support we added a new visitor to the back-end that can traverse every interface in the AST and generate the smart proxy framework classes shown in Figure 9. TAO's IDL compiler can be instructed to generate these smart proxy framework classes for every interface in an IDL file. Below, we described each of the classes shown in Figure 9:

• **Smart proxy factory,** which is provided by an application developer to create a custom smart proxy.

• **Proxy factory adapter,** which provides a singleton [4] container that manages the lifetime of the smart proxy factory registered with it. When a smart proxy factory is created by an application it registers itself automatically with this singleton. The proxy factory adapter takes ownership of this factory object and deletes it before the program terminates to ensure there are no memory leaks. Applications can also request an adapter to unregister its factory.

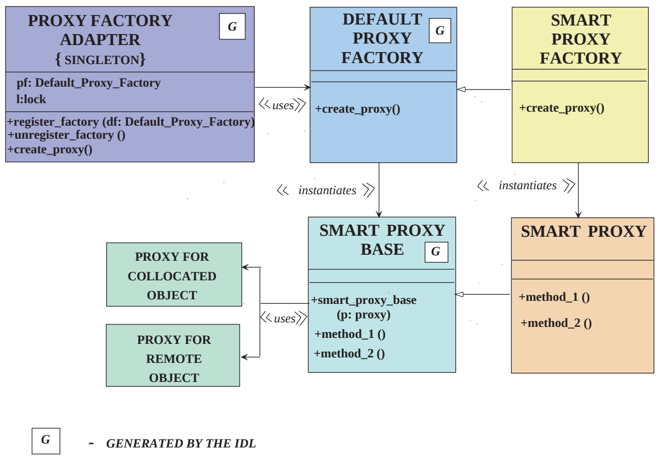


Figure 9: The Classes in TAO's Smart Proxy Framework

• **Default proxy factory,** which is the default factory object that returns the default proxy, *i.e.*, the stub that communicates with target objects. This factory registers itself with the adapter singleton during program initialization. It is deleted either (1) when it is replaced by another proxy factory or (2) when the program terminates and the singleton proxy factory adapter is destroyed; and

• **Interface-specific smart proxy base,** which is a class applications inherit from to define their custom smart proxies.

Below, we illustrate how to programming using TAO's smart proxies framework.

Smart proxy factory: This factory class is defined by application developers. It creates a smart proxy when a client application calls the standard CORBA `_narrow` operation. The smart proxy factory class must inherit from the default proxy factory class generated by TAO's IDL compiler. This design ensures the factory is registered automatically with the singleton [4] proxy factory adapter via a base class constructor. For example, a `TAO_Stock_Quoter_Default_Proxy_Factory` class is generated from the `Quoter` interface and can be inherited from as shown below:

```
class Smart_Quoter_Factory : public virtual
  TAO_Stock_Quoter_Default_Proxy_Factory
{
public:
  // This factory method will create the
  // smart proxy.
  virtual Stock::Quoter_ptr create_proxy
    (Stock::Quoter_ptr proxy);,

  // ...
};
```

Depending on policies set by applications, the scope of a smart proxy factory in TAO can be defined on a *per-interface* or *per-object* basis, as follows:

• **Per-interface:** With this policy the same smart proxy is used for all target objects associated with a particular IDL interface. When an object reference to a target object is obtained via `_narrow`, a smart proxy is created to act as the stub for all operation invocations on this object. This policy is the most transparent because after a smart proxy factory is instantiated for an interface `Foo`, all calls to `Foo::_narrow` will use this factory to create their smart proxies.

• **Per-object:** With this policy each target object can have a different smart proxy factory, which is less transparent but more flexible. After the first invocation on a target object the smart proxy factory is unregistered from the proxy factory adapter. Thus, unless a new smart proxy factory is installed explicitly for a new object, new object references to target objects of the same interface will use the default proxy. A new smart proxy factory could either be another instance of the one that was created earlier or a different smart proxy factory that will create another type of smart proxy for the target object.

The proxy factory adapter delegates the task of creating a smart proxy to the factory registered with it. By default, this factory creates the default proxy. The following factory method [4] shows how a smart proxy is created and how the default proxy is passed as the formal parameter to the factory method `create_proxy`:

```
Stock::Quoter_ptr
Smart_Quoter_Factory::create_proxy
  (Stock::Quoter_ptr proxy)
{
  // Verify the default proxy and use it
  // to create the new smart proxy.
  if (!CORBA::is_nil (proxy))
    proxy = new Stock_Quoter_Smart_Proxy (proxy);
  return proxy;
}
```

In the `_narrow` operation used to obtain the target object, a default proxy will be created. As shown in the method above, this default proxy is passed along to the `create_proxy` invocation on the `Smart_Quoter_Factor`. This factory method stores the default proxy in the smart proxy, which it can use to communicate with the target object.

Smart proxy class: TAO's IDL compiler generates a prototype of the smart proxy that inherits from the default proxy and the smart proxy base class. For example, the smart proxy base class generated for the `Quoter` interface is shown below:

```
// This class helps develop the smart proxy.
// Application-specific smart proxy classes
// inherit from this class.
class Stock_Quoter_Smart_Proxy_Base
  : public virtual Stock::Quoter,
    public virtual Smart_Proxy_Base
{
public:

  // Store the default proxy to perform the
  // actual work of passing the request
  // to the server.
  Stock_Quoter_Smart_Proxy_Base
    (Stock::Quoter_ptr proxy)
      : proxy_ (proxy) {}

  virtual CORBA::Long
  get_quote (CORBA::String stock_name)
    throw Invalid_Stock;

  virtual void
  log_quote (CORBA::Long quote);

protected:
  // Cache the original proxy reference.
  Stock::Quoter_var proxy_;
};
```

Applications can inherit from this class and implement methods that they want to override. For example, authentication can be added to validate the client before it receives the stock quote from the `Quoter` object, as follows:

```
class Stock_Quoter_Smart_Proxy :
  public Stock_Quoter_Smart_Proxy_Base
{
  // Smart proxy method.
  CORBA::Long
  Stock_Quoter_Smart_Proxy::get_quote
    (CORBA::String stock_quote)
    throw Invalid_Stock
  {
    CORBA::Long result = 0;

    try {
      // Authenticate the client using the
      // CORBA security service.
      result =
        security_service_->authenticate (key_);

      // Verify result, else throw exception.
      // ...

      // Call down to the default proxy to
      // send request to the target object.
      result =
        Quoter_Smart_Proxy_Base::get_quote
          (stock_name);
    } catch (Quoter::Invalid_Stock &) {
      // Deal with the exception caught ...
      return -1;
    }
    return result;
  }
  // ...
};
```

Client implementation: Below, we show a function that illustrates the per-interface smart proxy factory policy, where the client application explicitly creates one `Smart_Quoter_Factory` instance. This factory then creates a `Stock_Quoter_Smart_Proxy` object on each call to `_narrow` on the `Stock::Quoter` interface:

```
int main (int argc, char *argv[])
{
  // ... Initialize the ORB ...

  // Install the smart proxy factory for the
  // <Stock::Quoter> interface.  By default, the
  // factory created is per-interface.  If more
  // flexibility is needed, the factory can be
  // per-object, which is enabled by passing a 0
  // to its constructor.

  Smart_Quoter_Factory *quoter_factory
    = new Smart_Quoter_Factory (1);

  // ... Call the <current_quote> method for
  // various IORs ...
}
```

```
CORBA::Long current_quote (CORBA::ORB_ptr orb,
                           const char *ior)
{
  CORBA::Object_var obj =
    orb->string_to_object (ior);

  Stock::Quoter_var server
    = Stock::Quoter::_narrow (obj.in ());

  return server->get_quote ("ACME ORB");
}
```

As shown above, the only change required to existing client application code is to create an instance of `Smart_Quoter_Factory` before any calls to `_narrow` are made. Note that `Smart_Quoter_Factor` must be heap allocated since TAO's smart proxy framework classes is responsible for destroying this object. This design simplifies the tasks of (1) application developers, who need not manage smart proxy factory lifetimes at all and (2) smart proxy developers, who can manage the lifetime of their smart proxies more precisely.

Smart proxies can also be installed dynamically into an application via the Component Configurator pattern [10]. For example, the smart proxy factory can be stored in a dynamically linkable library (DLL). To accomplish this in TAO, we simply add an entry into the `svc.conf` configuration script to load this DLL on-demand:

```
dynamic Smart_Quoter_Factory Service_Object *
./Smart_Quoter_Factory :
  _make_Smart_Quoter_Factory() ""
```

As shown above, the smart proxy factory class resides in a DLL with the factory function entry point `_make_Smart_Quoter_Factory`, which is called automatically when the TAO ORB is initialized. This design allows smart proxies to be configured transparently without requiring *any* changes to existing client application implementations!

3.2 Using Portable Interceptors for Secure Transactions

Overview: As shown above, smart proxies can authenticate clients transparently via a trusted third-party. However, more powerful authentication mechanisms allow user information, such as credentials, to be sent for each request. To leverage these mechanisms in CORBA transparently, interceptors can be used to pass user information via the service context list that is tunneled with each GIOP request.

Below, we revise our stock quoter system so it uses interceptors to provide authentication information on a per-request basis via service context lists. In addition, we show how CORBA `Dynamic` module types can be used within interceptors to obtain additional request information, such as parameters, return values, and request ids.

Client interceptor: On the client, we use the following `send_request` interceptor hook to bundle authentication information into the service context.

```
class Secure_Client_Request_Interceptor : public
  PortableInterceptor::ClientRequestInterceptor
{
public:
  // ...
  void send_request
    (PortableInterceptor::ClientRequestInfo_ptr ri)
  {
    // The <password> is the authentication
    // information we send to the server.

    // Create the context to send the context
    // to the target.
    IOP::ServiceContext sc;
    sc.context_data.replace (strlen (password_),
                             strlen (password_),
                             password_,
                             1);
    // Add this context to service context list.
    ri->add_request_service_context (sc, 0);
  }
private:
  // Password we send to the server per-request.
  const char *password_;
}
```

A client request interceptor uses the `send_request` hook method above to create a new service context entry that adds a password into the existing request's service context list. The `ClientRequestInfo` object encapsulates the request's service context list so that it can be accessed by an interceptor. Next, we register the `Secure_Client_Request_Interceptor` instance with the client ORB, as follows:

```
  // This is code that would reside in a
  // concrete implementation of an
  // ORBInitializer::post_init() method, for
  // example.

  // Create and Install the client interceptor.
  PortableInterceptor::
  ClientRequestInterceptor_var
    interceptor = new
      Secure_Client_Request_Interceptor;

  // "info" is the ORBInitInfo argument
  // passed to this
  // ORBInitializer::post_init() method.
  info->add_client_request_interceptor
    (interceptor.in ());
```

This interceptor hook is one of the first interception points invoked on the client, as shown in Figure 3.

Server interceptor: On the server, we use an interceptor to verify the password sent via the service context list, as shown in the `receive_request` interceptor method below:

```
class Secure_Server_Request_Interceptor : public
  PortableInterceptor::ServerRequestInterceptor
{
public:
  void receive_request_service_contexts
    (PortableInterceptor::ServerRequestInfo_ptr ri)
  {
    IOP::ServiceContext *sc
      = ri->get_request_service_context (1);

    const char *buf = reinterpret_cast
      <const char *,
       sc->context_data.get_buffer ()>;

    // Verify the password.
    if (strcmp (sc->context_data.get_buffer (),
                "root") != 0)
      // throw exception ...
  }

  void receive_request
    (PortableInterceptor::ServerRequestInfo_ptr ri)
  {
    // Now check the parameters passed.
    if (strcmp (ri->operation (),
                "get_quote") == 0) {
      // Obtain parameter list.
      Dynamic::ParameterList paramlist
        = *ri->arguments ();
      CORBA::Long stock_quote;
      // Extract from the any.
      paramlist[0].argument >>= stock_quote;

      // if invalid stock quote throw exception
      ...
  }
}
```

The `receive_request_service_contexts` hook method obtains the service context from the service context list stored in the `ServerRequestInfo` object and verifies the password. The `receive_request` hook method then uses types defined in the ORB's `Dynamic` module to check the request parameters to ensure the quote is valid. The `Dynamic` module helps to build the parameter list on-demand and insert `any` variables that can be extracted by the interceptor as needed.

Finally, the `Secure_Server_Request_Interceptor` is registered with the server ORB, as shown below:

```
  // This is code that would reside in a
  // concrete implementation of an
  // ORBInitializer::post_init() method, for
  // example.

  // Create and install the server interceptor.
  PortableInterceptor::
  ServerRequestInterceptor_var
    interceptor = new
      Secure_Server_Request_Interceptor;

  // "info" is the ORBInitInfo argument
  // passed to this
  // ORBInitializer::post_init() method.
  info->add_server_request_interceptor
    (interceptor.in ());
```

After this interceptor has been installed, it will be invoked at all interception points along the server's invocation path. The particular point that will call the `receive_request` method is after parameter demarshaling, but just before the POA makes the upcall to the servant. For example, we could first authenticate and then use the information passed to set policies for the access rights granted to the client for a particular target object.

4 Key Design Challenges and Pattern-based Resolutions

In this section, we explore how smart proxies and interceptors are implemented in TAO. To clarify and generalize our approach, the discussion below focuses on the patterns [4] we applied to resolve the key design challenges faced during our development process.

4.1 Smart Proxy Design Challenges and Resolutions

As mentioned in Section 2.1, the goal of using smart proxies is to change/add behaviors to existing programs with minimal modifications to client applications. Below, we discuss the key design challenges we faced while refactoring TAO's existing stub architecture to support smart proxies.

4.1.1 Challenge: Providing Flexible Support for Smart Proxies

Context: The proxy framework generated by TAO's IDL compiler should allow applications to use customized proxies transparently. For example, changes to client applications that use customized proxies must be localized. In particular, developers should be able to install customized proxies with little or no change to client application code.

Problem: TAO's original IDL compiler generated only fixed default proxies. In particular, the `_narrow` operation it generated for each interface returned a default proxy. If developers require more flexibility, however, the `_narrow` operation must be able to return either an IDL-generated default proxy or a custom smart proxy.

Since the `_narrow` operation is generated by TAO's IDL compiler as part of the client's stub it is not possible to modify this method externally from a client application. Moreover, since fixed default stubs were generated any changes required manually modifying the IDL-generated code. Clearly, this solution was inflexible and had to be solved at the stub-generation level.

Solution → Apply the Factory Method, Adapter, and Singleton patterns: We applied these design patterns [4] in TAO's smart proxy framework to provide the necessary flexibility to create different types of proxies transparently in TAO's IDL-generated code, as follows:

- The Factory Method pattern defers instantiation of various types of meta-objects to subclasses.
- The Adapter pattern provides a higher level of abstraction for TAO's proxy factories and to delegate creation requests to the appropriate factory.
- The Singleton pattern makes the proxy factory adapter a global access point for factory registration from program initialization to termination.

Figure 10 illustrates how we applied these three patterns in TAO to provide flexible support for smart proxies. By using

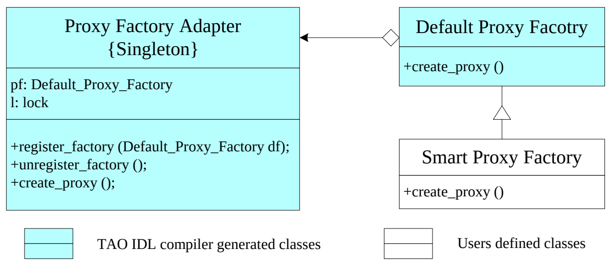


Figure 10: Applying Patterns to Provide Flexible Support for Smart Proxies

these patterns, applications can obtain either the default IDL-generated proxy or a smart proxy without changing existing code manually. For example, after an application registers a per-interface smart proxy factory, the `_narrow` operation call will create the appropriate proxy automatically.

4.1.2 Challenge: Treating Remote and Collocated Smart Proxies Uniformly

Context: A target object can be either remote or it can be collocated in the client's address space [16]. TAO provides customized meta-objects called *collocated proxies* to optimize performance for collocated objects. Smart proxies should provide similar functionality to collocated and remote proxies since the ability to differentiate remote and collocated smart proxies provides developers with greater flexibility.

Problem: Depending on where a target object resides, a developer may or may not wish to invoke the smart proxy installed for the object. For example, a developer may not want to cache operation results in a collocated smart proxy because these calls are already resolved locally. Originally, TAO treated the generation of collocated stubs as a special case and if smart proxies were installed they would supercede the default stubs, even if the stubs were collocated.

Ignoring collocation optimizations, however, may cause unnecessary waste by trying to optimize a bottleneck that does not exist. Therefore, it is necessary to distinguish the remote and collocated case to take full advantage of this construct and avoid unnecessary waste of system resources, such as memory and CPU cycles. In addition, smart proxies must (1) provide applications with the same interface as default proxies and (2) be able to call down to the default proxy to communicate with remote target objects.

Solution → Apply the Composite pattern: The Composite pattern [4] supports part/whole relationships and allows all objects in such composite structures to be processed uniformly. We applied the Composite pattern to TAO to provide a uniform view among different proxies available to clients. As shown in Figure 11, in this design (1) smart proxy classes inherit from

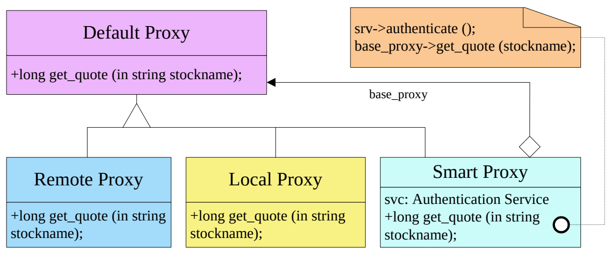

Default Proxy
+long get_quote (in string stockname);
srv->authenticate ();
base_proxy->get_quote (stockname);
base_proxy
Remote Proxy
+long get_quote (in string stockname);
Local Proxy
+long get_quote (in string stockname);
Smart Proxy
svc: Authentication Service
+long get_quote (in string stockname);

Figure 11: Applying the Composite Pattern to TAO's Smart Proxy Design

the default proxy and (2) also store a pointer to the default proxy to make invocations to target object. Collocated and remote proxies are children of the default proxy. Thus, smart proxies can make calls to the remote or collocated proxy transparently, while providing the same application interface as the default proxies.

4.2 Interceptor Design Challenges and Resolutions

As discussed in Section 2.2, interceptors can extend the behavior of CORBA operations with minimal changes to client and server applications. In this section, we discuss the key design challenges faced while enhancing TAO's existing invocation architecture to support interceptors.

4.2.1 Challenge: Making Information Retrieval Possible Per-Operation

Context: Request interceptor hook methods are invoked at different interception points along the invocation path. These interceptors must be able to (1) verify and audit information being passed to the target object as the invocation continues and (2) potentially terminate the invocation before it reaches the target object.

Problem: An ORB must provide information in response to interceptor queries. This information may be operation-specific and even temporal. For example, the result of an operation may be available only after the POA makes an upcall to a servant and the operation executes.

An ORB must therefore have a generic way to access operation-level information and disclose this information to interceptors that are invoked at ORB-mediated interception points. Originally, TAO did not maintain this information to avoid degrading the normal execution of the invocation in situations where this information was not required by applications. However, TAO's original design made it hard for applications to influence invocation behavior.

Solution → Generation of nested RequestInfo classes for each interface operation: To provide invocation information dynamically and efficiently, we modified TAO's IDL compiler to generate `RequestInfo` classes for each operation. `RequestInfo` classes are instantiated for each operation invocation and passed to the interceptors during the invocation. Thus, interceptors can access operation-related information, as shown in Figure 12. Every operation in an IDL interface

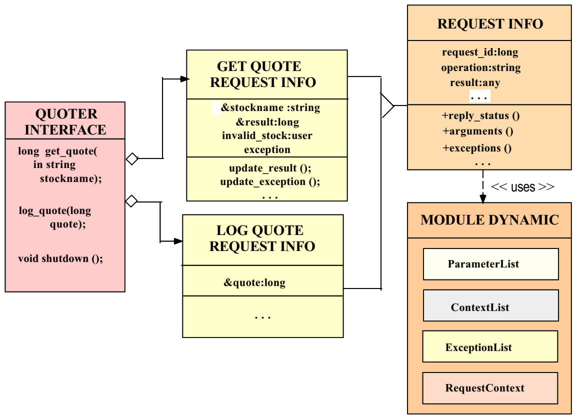


Figure 12: TAO's Portable Interceptor Design

may have different formal parameters, result types, and user exceptions. To minimize the overhead of copying multiple arguments and the return value of the upcall, we only store a reference, rather than a copy of the parameters, results, and exceptions.

We added TAO-specific methods to each `RequestInfo` class and used these methods internally to update the result and the exception thrown, rather than instantiating a new `RequestInfo` class before every interception point is called. For instance, the result of an operation is obtained only after the POA makes the upcall and the client receives a reply. At this point, the client can verify the result in the `receive_reply` interceptor hook by querying the `RequestInfo` object, making it necessary to update the result before this interception point is invoked. Thus, temporal information can also be propagated to interceptors.

4.2.2 Challenge: Avoiding Gratuitous Waste Constructing RequestInfos

Context: Interceptors can access any request-related information. Their interface must therefore be sufficiently general to incorporate any type of data. In CORBA, `any` is a generic type that can hold information of any other types, which are stored using type/value tuples.

Problem: In general, not all interceptors installed in an ORB are interested in handling all information, or even all operations. For example, security-related interceptors may not be interested in what operation is being invoked, but only want to know the contents of the service context list. Likewise, an auditing interceptor may only be interested in the parameters of certain operations of certain objects, while ignoring others altogether.

Although CORBA's `any` type is flexible, it is less efficient and more resource consumptive than other common CORBA data types, such as `long` or `struct`. We need to avoid the overhead of `any` insertion operators if installed interceptors are not interested in certain operation information. There is no way, however, to predict what interceptors will be interested in *a priori*.

Solution → On-demand creation of operation information: To avoid unnecessary waste of resources, we applied the Lazy Initialization pattern [17] to make sure the operation information is only inserted into `any` objects the *first time* a related interface is accessed by an interceptor via its `RequestInfo`-derived interface. This design ensures that pertinent information in `RequestInfo`-derived objects will only be created if an interceptor is interested in the information. In TAO, we retrieve this information via types defined in the CORBA `Dynamic` module.

The `Dynamic` module defines the collocation of request parameters, results, and exceptions in `any` in a sequence of structures that an application interceptor can extract and use. In TAO, methods returning Dynamic objects are implemented to minimize the gratuitous waste of storing all information `de facto` into lists of `any`s as shown in Figure 13. In particular, this information is inserted into `any`s only when queried, which occurs just once. Subsequent queries simply return the `any` variables created previously. Thus, unless an interceptor needs to query a particular piece of request information, it incurs no additional overhead. This optimization is targeted for the common case where interceptors are used to pass service contexts.

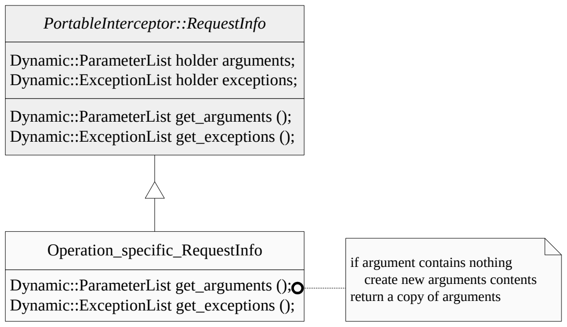

PortableInterceptor::RequestInfo
Dynamic::ParameterList holder arguments;
Dynamic::ExceptionList holder exceptions;
Dynamic::ParameterList get_arguments ();
Dynamic::ExceptionList get_exceptions ();
Operation_specific_RequestInfo
Dynamic::ParameterList get_arguments ();
Dynamic::ExceptionList get_exceptions ();
if argument contains nothing
create new arguments contents
return a copy of arguments

Figure 13: TAO applies Lazy Initialization building Dynamic objects in RequestInfo

4.2.3 Challenge: Implementing Time and Space Efficient Flow Stacks

Context: The Portable Interceptor specification defines *general flow rules* to which a portable interceptor implementation should adhere. These rules ensure that only interceptors invoked successfully from a starting interception point will ever be invoked at an ending interception point. Conceptually, interceptors are pushed on to a stack if invoked successfully in a starting interception point and popped off that stack when they invoked at ending interception points.

Problem: To implement the semantics dictated by CORBA's general flow rules, some type of stack implementation is needed. However, implementing a *flow* stack with a general-purpose stack container class, such as the one in the standard C++ library [18], has the following problems:

• **Time overhead:** The stack implementation may incur non-trivial performance overhead if it allocates space off of the heap dynamically for each interceptor or interceptor reference pushed onto the stack. Dynamic memory is particularly problematic for real-time applications.

• **Space overhead:** The stack implementation itself adds to the ORB footprint since a template must be instantiated for each type of request interceptor, *i.e.*, client or server request interceptors. Moreover, other auxiliary templates may need to be instantiated for internal stack support code. Not only does this increase the static footprint of the ORB, but it also increases run-time ORB memory requirements, which may be unacceptable for embedded applications.

In addition to inherent problems with real stack implementations detailed above, another common problem can occur. Since interceptors are invoked during a request, they are in the critical path. This means that interceptor support code, such as a flow stack, can have an adverse affect on performance if that support is not implemented efficiently. In particular, adding locking mechanisms in the flow of a request can degrade performance since threads waiting for a lock can block. The act of acquiring and releasing the lock also imposes further delays.

Solution $\rightarrow$ Apply optimization principle patterns: Optimization principle patterns [19] define a set of principles that can be applied to improve performance in various ways. To implement time and space efficient flow stacks, heap allocations must be minimized to avoid degrading performance and increasing footprint. Both can be avoided by taking advantage of *pre-computed* resources and the properties associated with them.

As dictated by the Portable Interceptor specification, interceptors are registered with the ORB when the ORB is bootstrapped, *i.e.*, during the initial `CORBA::ORB_init` call. This means that storage for the interceptors will already have been allocated by the time the interceptors are invoked so there should ideally be no need for additional allocations at a later point in time.

By keeping the order with which the interceptors are stored unchanged for the lifetime of the ORB, it is possible to implement highly efficient stack *push* and *pop* operations. Interceptors will always be pushed on to the stack with the same relative ordering they are stored in the ORB. This property ensures that the number of elements on the stack will be equal to the ORB storage location of the last interceptor pushed on to the stack. Hence, the general flow rule semantics can be implemented using a *logical* flow stack.

Applying the solution to TAO: TAO stores pointers to registered interceptors in a pre-allocated array, which avoids increased footprint and run-time memory requirements. Rather than having to instantiate a stack for each type of interceptor (*i.e.*, client and server request interceptors), a single array for each type of request interceptor is created. The order in which interceptors are stored in the array remains unchanged for the lifetime of the ORB. Thus, *push* and *pop* operations can be implemented by simply incrementing and decrementing a variable, respectively, as illustrated in Figure 14.

The following example presents a scenario that illustrates how TAO's logical flow stacks are implemented:

1. Three request interceptors are registered when the ORB is initialized. Specifically, the `CORBA::ORB_init` method invokes all ORB initializers registered by the application.

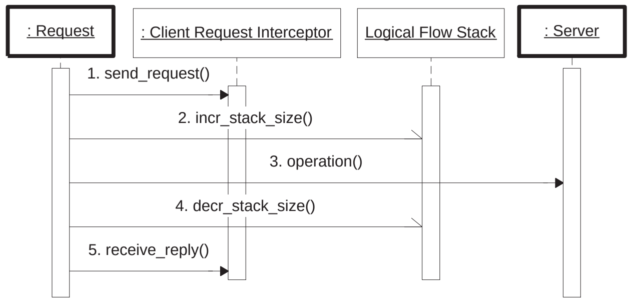


Figure 14: An Efficient Flow Stack Implementation

Those ORB initializers then register the interceptors by using the appropriate methods in the `ORBInitInfo` argument passed to the ORB initializer by the `CORBA::ORB_init` method. An example of this interceptor registration code follows:

```
// Code that would reside in a
// concrete implementation of an
// ORBInitializer::post_init() method, for
// example.

// Create and install a client interceptor.
PortableInterceptor::
ClientRequestInterceptor_var
  interceptor = new
    Secure_Client_Request_Interceptor;

// "info" is the ORBInitInfo argument.
info->add_client_request_interceptor
  (interceptor.in ());
```

2. Two interceptors are successfully invoked at a *starting* interception point during a request. This corresponds to step 1 in Figure 14.

3. Each successful request interceptor invocation increments the stack size by one, which results in a stack size of two. Stack element `one` corresponds to request interceptor `one` as stored in the ORB's interceptor array. Similarly, stack element `two` corresponds to interceptor `two` in the ORB's interceptor array. Again, a *logical* stack is in use here. This corresponds to step 2 in Figure 14.

4. An *ending* interception point is invoked.

5. Within the ending interception point, each of the interceptors in the logical stack is invoked. Prior to invoking each interceptor, the stack size is decreased by one (step 4 in Figure 14), effectively popping an interceptor off of the logical flow stack. Since only the first two interceptors were pushed on to the stack, only the first two of the three interceptors will be invoked (step 5 in Figure 14) in the ending interception point and the third interceptor will never be invoked.

TAO's logical flow stack implementation allows the CORBA general flow rule semantics to be implemented efficiently and with minimal impact on ORB footprint. These benefits arise from the fact that flow stack storage is preallocated prior to the first use of the flow stack. In addition, the TAO implementation ensures the order of the interceptors stored in the ORB's interceptor array remains unchanged for the lifetime of the ORB.

One other aspect of this implementation is the fact that it is *not* necessary to acquire a lock to prevent other threads from modifying the logical stack. Only one thread ever services a request at a given time. Thus, there is no need to implement a locking mechanism for the logical stack, in which case additional overhead is not incurred.

5 Empirical Benchmarking Results

Developers of distributed applications must often make tradeoffs between time/space overhead and flexibility. Selecting which meta-programming mechanism to use, *e.g.*, smart proxies or interceptors, is an example of this tradeoff. This section presents benchmarking results that quantify the time/space overhead and tradeoffs of using smart proxies and portable interceptors.

5.1 Overview of the Testbed Environment and Benchmarks

The experiments were conducted using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor UltraSPARC-2s running SunOS 5.7. Each UltraSPARC-2 contains 2 168 MHz CPUs with a 1 Megabyte cache per-CPU, 256 Mbytes of RAM, and an ENI-155s-MF ATM adapter card that supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The experimental testbed is shown in Figure 15. The benchmarking programs were compiled using the Sun CC 5.0 compiler with all optimizations enabled. We conducted two different benchmarks: one measured the performance of smart proxies and the other the performance of interceptors.

5.1.1 Smart Proxy Results

The overhead of calling an operation via a smart proxy is equivalent to calling the default proxy, *i.e.*, it is the cost of a local virtual method call. Therefore, we designed our smart proxy benchmark to show how performance can be improved if smart proxies are used as a cache to minimize the number of remote operations. Here is the IDL interface we used for this test:

```
interface Broadway_Show
{
```

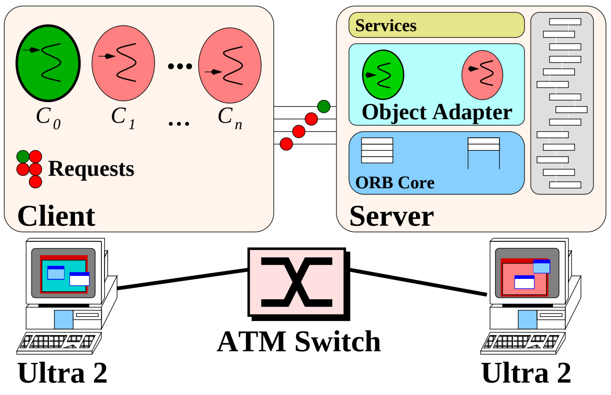


Figure 15: Testbed for Meta-programming Mechanism Benchmarks

```
  // Get the prices for the box
  // seats of the Broadway show.
  short box_prices ();

  // Order tickets.
  long order_tickets (in short number);
};
```

The servant in the test is a virtual box office that allows clients to purchase tickets to Broadway shows. A client can query the prices of box seats and if they are within a price range, it buys them. Thus, the client normally makes two invocations: (1) `box_prices` and (2) `order_tickets` if the prices are reasonable. By default, every time a client enquires about ticket prices, a remote invocation occurs.

We can minimize overhead significantly by using a smart proxy that makes just one remote invocation and then caches the result and reuses it when subsequent enquiries occur. This caching improves the performance significantly, as shown in Figure 16. This figure illustrates that omitting unnecessary

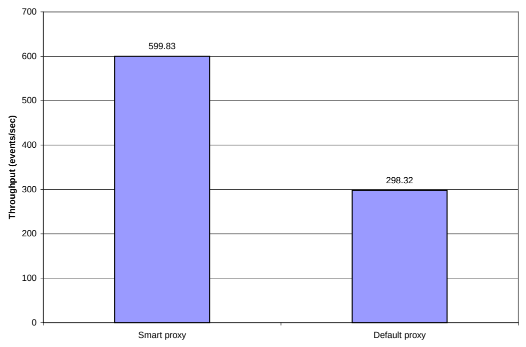


Figure 16: Performance Improvement Using a Smart Proxy to Cache Information

remote operation calls improve the performance by ∼130%, even over a high-speed ATM network.

5.1.2 Portable Interceptor Results

Our portable interceptor benchmarks quantify the cost of supporting and using interceptors in TAO. Moreover, these tests quantified the costs of individual interceptor features, such as accessing a parameter list and accessing a service context list. In the benchmark program, the following three IDL operations were defined in the `Secure_Vault` interface:

```
interface Secure_Vault
{
  exception Invalid {};

  struct Record { long check_num; long amount; };

  // No args/exceptions operation.
  short ready ();

  // Throws a user exception.
  void authenticate (in string user)
    raises (Invalid);

  // updates a struct and returns a count.
  long update_records (in long id,
                       in Record val);
};
```

Each operation takes a different number and different length of parameters and return values. Moreover, the `authenticate` operation throws a user exception, whereas the other two do not. This diversity allowed us to measure the cost of preparing different types of generic information required by interceptors.

The interceptor benchmarks were run using the five different configurations summarized below:

1. No interceptor support: In this configuration, interceptor support was disabled completely in the ORB, which measured TAO's baseline performance.

2. No interceptor installed: This time the ORB was compiled with interceptor support, although the test was performed without installing an interceptor into the ORB. This configuration measures the performance penalty applications must pay for the potential of flexibility.

3. No-op interceptor installed: This configuration uses a no-op interceptor to measure the cost of invoking interceptors.

4. Accessing the service context list: The interceptor installed in this configuration manipulates the GIOP request's `ServiceContextList`. On the client, a request interceptor creates a new `ServiceContext` containing an encapsulated password string of 7 bytes and inserts the service context object into the `ServiceContextList` of the invocation using the `RequestInfo` interface. On the server, a different request interceptor performs the reverse operation by (1) extracting the password string from the `ServiceContextList` using the `RequestInfo` interface and (2) examining the password via a string comparison.

5. Accessing Dynamic information: TAO implements the `Dynamic` module types in request/reply operations, such as parameters, results and exception list of an invocation, by creating these information on-demand. The interceptor installed in this configuration accesses the dynamic information of the operations by checking their parameters and return values.

Figure 17 shows the cost of supporting and using these various features and configurations in interceptors. In the first con-

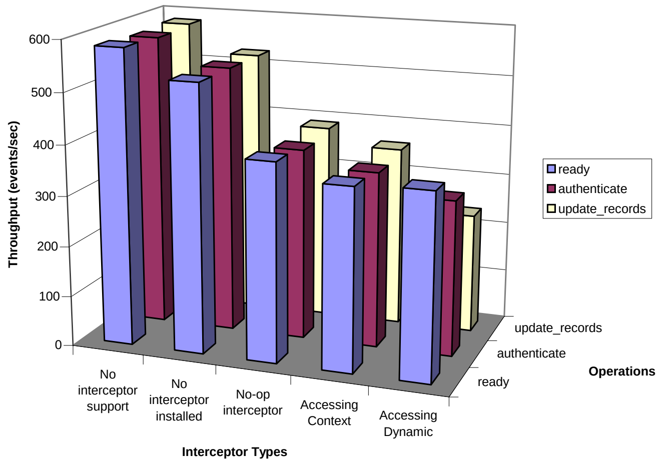


Figure 17: Cost of Using Various Interceptor Features

figuration (no interceptor support), all three measured operations perform similarly because there is no significant difference between the information these operations exchange. The results are similar for the second configuration, which added interceptor support to the ORB but without installing any interceptors. There is only a ∼9% performance penalty for using the ORB with interceptor support.

The no-op interceptor provide the baseline cost of invoking an interceptor. There is ∼26% of performance penalty compared to not installing the interceptor due to invocations of interception points on every operation invocation. As shown in Figure 17, however, all three operations reveal similar performance characteristics, regardless of the number and size of their parameters and return values.

Similar performance degradation is also observed for interceptors that access the `ServiceContextList`. This configuration measures the cost of adding and extracting a short string from the `ServiceContext`. Again, all three operations experience ∼8% degradation in performance compared to using the no-op interceptor.

The interceptor that access the `Dynamic` module types, however, demonstrates more diversity in performance degradation among the three operations we tested. There are ∼7%, ∼19%, ∼and 40% performance hits to the `ready`, `authenticate`, and `update_record` operations, respectively, compared with no-op interceptor configuration. The performance penalty comes not only from the accessing parameters using the `Dynamic` module types, but also from the on-demand creation of the dynamic information. The results show that the preparation of `Dynamic` module types are expensive, which justifies our decision not to create them if they are not accessed by interceptors.

5.2 Memory Footprint Results

TAO is an open-source ORB that is used for real-time and embedded systems with memory constraints. Therefore, smart proxies and interceptors can be conditionally compiled in or out at ORB compile-time. To measure the memory increment necessary to support smart proxies and interceptors, we compiled the `Secure_Vault` IDL interface shown above with three different operations using the following configurations:

1. Interceptors and smart proxies disabled.
2. Interceptors and the smart proxies both enabled;
3. Interceptors enabled but smart proxies disabled, which is the default configuration in TAO; and
4. Interceptors disabled and smart proxies enabled.

Table 1 shows the resulting sizes for different configurations. Not counting the application-specific proxy and factory

Supporting Config.	**Stub size (KB)**	**% Inc.**	**Skeleton size (KB)**	**% Inc.**
Neither	1,288	0	1,277	0
Smart proxies	1,321	2.5	1,277	0
Interceptors	1,479	14.8	1,485	16.3
Both	1,517	17.8	1,489	16.6

Table 1: Footprint Comparison for Smart Proxies and Interceptors

method, smart proxies increase TAO's client memory footprint by ∼2.5%. In contrast, interceptors require ∼15% extra footprint to handle on-demand creation of parameters lists, exceptions list, etc.

We also performed the same test using the OMG *Minimum CORBA* configuration [20], which defines a subset of the complete ORB CORBA specification to reduce embedded system memory footprints. By default, TAO's Minimum CORBA footprint is less than 1 MB. To determine the footprint growth when smart proxies and/or interceptors are used, we measured the size of the ORB again using the same IDL interface, as shown in Table 2: The footprint increase for TAO's smart

Supporting Config.	Stub size (KB)	% Inc.	Skeleton size (KB)	% Inc.
Neither	923	0	896	0
Smart proxies	974	5.5	896	0
Interceptors	1,115	20.7	1,104	23.1
Both	1,148	24.3	1,105	23.2

Table 2: Footprint Comparison for Smart Proxies and Interceptors in TAO's Minimum CORBA Configuration

proxies in this configuration is 5.55% and the support for interceptors causes a significant 20-23% increment. These results are not surprising since both these meta-programming features are new and have not yet been optimized for TAO's Minimum CORBA configuration.

In general, the results in this section show that CORBA meta-programming mechanisms can provide developers with significant improvements in functionality, performance, and convenience without drastic changes to existing application software. Depending on which features are used, however, developers need to consider the affect of time and space overhead.

6 Related Work

CORBA is increasingly being adopted as the middleware of choice for a wide-range of distributed applications and systems. As systems evolve, new features/services will be added to the system. Smart proxies and interceptors are good ways to adapt existing applications to take advantage of these new features. The following work on middleware technologies is related to our research.

QuO: The *Quality Objects* (QuO) distributed object middleware is developed at BBN Technologies [21] by applying Aspect-Oriented Programming (AOP) [22] techniques to adaptive network applications. 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; and

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.

QuO achieves this functionality via customized smart proxies, called *delegates*, and embedded MOP interfaces within the proxies. However, their framework does not allow users to install user-defined proxies and the MOP interfaces are specifically designed for QoS purpose.

Orbix filters: Orbix defines the concept of filters, which are an interceptor mechanism based on the concept of "flexible bindings" [23]. By deriving from a predefined base class, developers can intercept events. Common events include client-initiated transmission and arrival of remote operations, as well as the object implementation-initiated transmission and arrival of replies. Developers can choose whether to intercept the request or result before or after marshaling. Orbix programmers can leverage the same filtering mechanism to build multithreaded servers [24, 25, 26].

dynamicTAO: The dynamicTAO reflective ORB [27] supports interceptors for monitoring and security. Particular interceptor implementations are loaded into dynamicTAO using the Component Configurator pattern [10]. Using component configurators to install interceptors in dynamicTAO allows applications to exchange monitoring and security strategies at run-time. Moreover, there are extensive use of reflective programming technique in dynamicTAO to determine the module the ORB requires.

Fault-tolerant ORB frameworks: Interceptors have been applied in a number of fault-tolerant ORB frameworks such as the Eternal system [28]. Eternal intercepts system calls made by clients through the lower-level I/O subsystem and maps these system calls to a reliable multicast subsystem. Eternal does not modify the ORB or the CORBA language mapping, thereby ensuring the transparency of fault tolerance from applications.

COM interceptors: Hunt and Scott [29] describe how to implement interceptors in COM. The concept they use to implement interceptors is similar to TAO's collocated stub [16]. This technique uses alternative wrappers around the object implementation to masquerade as operation targets, which are similar to TAO's smart proxies.

7 Concluding Remarks

Distributed object computing (DOC) middleware has been applied widely to domains ranging from telecommunications to aerospace, process automation, and e-commerce. DOC middleware shields developers from many distribution challenges and allows applications to invoke operations on target objects efficiently without concern for their location, programming language, OS platform, communication protocols and interconnects, and hardware [30]. Historically, however, many DOC middleware solutions have tightly coupled interfaces and implementations, which makes it hard to adapt to requirement or environment changes that occur late in an application's life-cycle, *i.e.*, during deployment and/or at run-time.

Meta-programming mechanisms are techniques that help increase the flexibility and adaptability of applications, without degrading performance significantly. This paper describes two meta-programming mechanisms–*smart proxies* and *interceptors*– that we added recently to TAO, is an implementation of CORBA that is targeted for applications with high-performance and real-time QoS requirements. These two mechanisms allow CORBA applications to adapt to changing requirements or environmental conditions that occur late in an application's life-cycle without requiring obtrusive changes in existing software.

Based on our experience using smart proxies and interceptors to develop TAO applications, we have observed the following tradeoffs and limitations with smart proxies and interceptors:

Performance: Interceptors incur more overhead than smart proxies because they influence the processing of operations at multiple points along the invocation path. The portable interceptor results in Section 5.1.2 illustrate the overhead of supporting interceptors and the run-time costs of specific interceptor features.

In general, smart proxies perform better and consume less memory than interceptors. The smart proxy results in Section 5.1.1 show the circumstances where using smart proxies can improve performance. Even thought there is an extra layer of indirection, the overall performance can be improved by removing the gratuitous overhead of unnecessary remote invocations.

Generality: Interceptors can be applied to either servers or clients and can access operation-specific information. Therefore, they provide an effective meta-programming mechanism to handle advanced features, such as authentication and authorization, transparently end-to-end. In contrast, smart proxies only apply to specific interfaces accessed by clients. In particular, smart proxies can only influence the behavior at the beginning of an invocation.

Portability: Smart proxies are not currently part of the CORBA standard. Although many ORBs provide smart proxies as extensions, this feature is not portable. There is, however, a Portable Interceptors specification [7] that is being ratified by the OMG.

All the source code, documentation, and tests for TAO are open-source and can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

Acknowledgements

Thanks to Brian Wallis <brian.wallis@ot.com.au> for helping with the design of TAO's smart proxy interface.

References

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

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

[3] 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.

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

[5] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom, "Flick: A Flexible, Optimizing IDL Compiler," in *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, (Las Vegas, NV), ACM, June 1997.

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

[7] Adiron, LLC, *et al.*, *Portable Interceptor Working Draft – Joint Revised Submission*. Object Management Group, OMG Document orbos/99-10-01 ed., October 1999.

[8] Object Management Group, *Security Service 1.8 Specification*, OMG Document security/00-11-03 ed., November 2000.

[9] O. Othman, C. O'Ryan, and D. C. Schmidt, "The Design and Performance of an Adaptive CORBA Load Balancing Service," *IEEE Distributed Systems Online*, vol. 1, December 2000.

[10] 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.

[11] 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.

[12] T. Nakajima, "Dynamic Transport Protocol Selection in a CORBA System," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.

[13] 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.

[14] W. W. Ho and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, pp. 375–390, Apr. 1991.

[15] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, and M. Kircher, "Applying C++, Patterns, and Components to Develop an IDL Compiler for CORBA AMI Callbacks," *C++ Report*, vol. 12, Mar. 2000.

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

[17] K. Beck, *Smalltalk Best Practice Patterns*. Englewood Cliffs, NJ: Prentice-Hall, 1997.

[18] M. H. Austern, *Generic Programming and the STL*. Reading, MA: Addison-Wesley, 1999.

[19] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Using Principle Patterns to Optimize Real-time ORBs," *Concurrency Magazine*, vol. 8, no. 1, 2000.

[20] Object Management Group, *Minimum CORBA - Joint Revised Submission*, OMG Document orbos/98-08-04 ed., August 1998.

[21] 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.

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

[23] M. Shapiro, "Flexible Bindings for Fine-Grain, Distributed Objects," Tech. Rep. Rapport de recherche INRIA 2007, INRIA, Aug. 1993.

[24] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Object," *C++ Report*, vol. 8, July 1996.

[25] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool," *C++ Report*, vol. 8, April 1996.

[26] D. Schmidt and S. Vinoski, "Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread-per-Request," *C++ Report*, vol. 8, February 1996.

[27] 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.

[28] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, vol. 32, pp. 64–68, July 1999.

[29] 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.

[30] S. Vinoski, "New Features for CORBA 3.0," *Communications of the ACM*, vol. 41, pp. 44–52, October 1998.