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

Nanbor Wang   Kirthika Parameswaran   Douglas Schmidt

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

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

This paper has been submitted to the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), San Antonio, Jan/Feb, 2001.

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 drastically changing existing software.

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 [1]. The magnitude of these complexities, combined with increasing time-to-market pressures, make it increasingly impractical to develop distributed applications 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, multi-threading, and synchronization; and

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

In the case of standards-based DOC middleware, such as CORBA [2], these capabilities are realized via an open specification process. The resulting products can interoperate across many OS/network platforms and many programming languages [3].

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 changing requirements or environmental conditions that occur late in an application's lifecycle, *i.e.*, during deployment and/or at run-time. In this paper, therefore, we describe and qualitatively compare the following two meta-programming mechanisms:

- **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 transparently monitor or modify the behavior of the invocation.

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 [4], 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 two meta-programming mechanisms–*smart proxies* and *interceptors*; Section 3 presents several programming examples illustrating the use of smart proxies and interceptors; Section 4 describe the patterns that (1) guide the architecture of TAO's smart proxy and interceptor mechanisms and (2) resolve 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 [5] and marshal operation information and data type parameters into a common request format. Likewise, CORBA skeletons implement the *Adapter* pattern [5] and demarshal the common request format back into typed parameters and operation information that are meaningful to a server application.

Stubs and skeletons can be generated automatically from schemas defined using some type of interface definition language (IDL). For example, an CORBA IDL compiler transforms 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, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations [6].

Traditionally, the stubs and skeletons generated by an OMG IDL compiler are *fixed, i.e.,* the code emitted by the IDL compiler is determined at translation time. This design shields application developers from tedious 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 their requirements or environmental conditions, such as the following:

- 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 traditional CORBA middleware with 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 applications can use 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* [7], as shown in Figure 1. As shown in this figure, a stub implemented as a meta-object

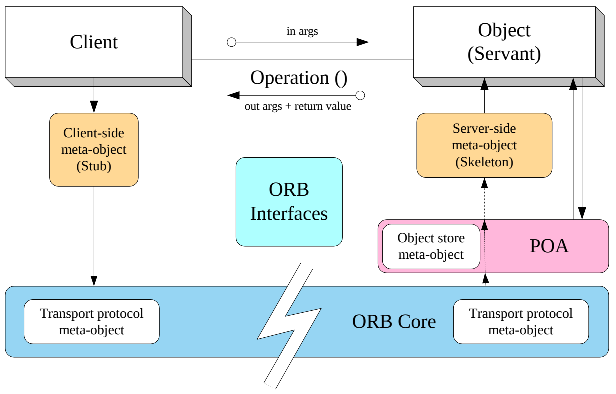

Client
in args
Object (Servant)
Operation ()
out args + return value
Client-side meta-object (Stub)
Server-side meta-object (Skeleton)
ORB Interfaces
Object store meta-object
POA
Transport protocol meta-object
ORB Core
Transport protocol meta-object

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

can act in conjunction with other interception meta-objects to transform and transmit a client operation invocation across the network to a server. Various meta-objects on the server's request processing path can then reconstruct the original operation invocation. Server meta-object's are also responsible for sending the result(s) of the invocation back to the client.

If all remote operation invocations are passed through meta-objects, certain aspects of application and middleware behavior can adapt transparently when system requirements and environmental conditions change by simply modifying the behavior of the meta-objects. To change meta-object behaviors, the DOC middleware can either (1) provide modified meta-objects for the client or (2) embed *hooks* implementing a *meta-object protocol* (MOP) in the meta-objects to invoke operations that provide customized behavior for meta-objects based on the data passed by the meta-objects. In the context of CORBA, customized meta-objects are called *smart proxies* and other meta-objects that implement the MOP are called *interceptors*.

2.1 Overview of Smart Proxies

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

To support these capabilities *without* modifying existing client code, applications must be able to override the default stub implementations. These application-defined stubs are called *smart proxies*. In ORBs with smart proxy support, a stub is a meta-object that acts as the proxy to the "real" target object.

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

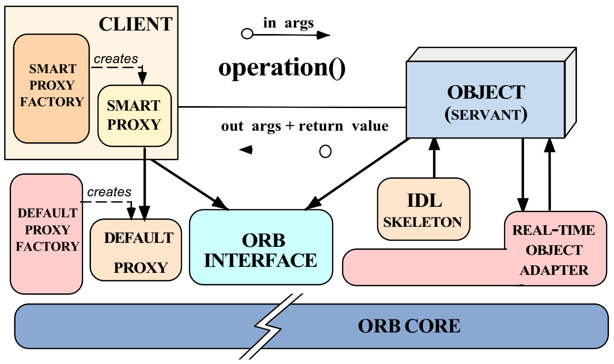


Figure 2: TAO's Smart Proxy Model

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 the `_narrow` operation of an interface is invoked. 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.

The registration of smart proxies are not yet standardized in CORBA, though many ORBs support this feature. In ORBs with smart proxy support they can be used as pluggable meta-objects that are cognizant of their target objects. This design allows developers to modify the behavior of interfaces without re-implementing client applications or target objects.

2.2 Overview of Interceptors

As described above, smart proxies are a meta-programming mechanism that increases the flexibility of client applications. Are another common meta-programming mechanism used in DOC middleware are *interceptors*, which help increase the flexibility of both client and server applications. In CORBA, interceptors are meta-objects that stubs, skeletons, and certain points in the end-to-end operation invocation path can invoke at predefined "interception points."

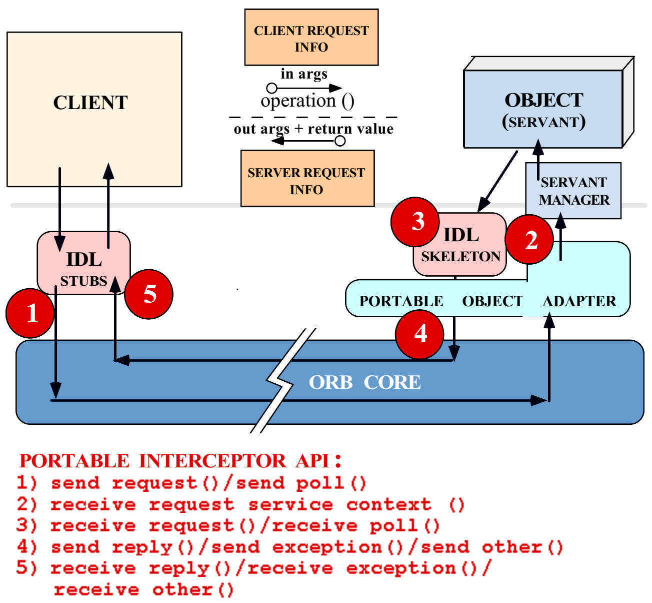
The interceptor specification defined prior to CORBA 2.3.1 was not portable. Therefore, the interceptors discussed in this paper are based on the "Portable Interceptors" specification [8], which currently being ratified by the OMG. Several types of interceptors are defined in the CORBA Portable Interceptor specification. They can be divided into two major categories: (1) *request interceptors* that deal with operation invocations and (2) *IOR interceptors* that insert information into interoperable object references (IORs), both of which are described below.

2.2.1 Request Interceptors

The request interception points defined in the Portable Interceptor specification are shown in Figure 3. Interception points occur in multiple parts of the end-to-end invocation path, including when an object is created, 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.

A server ORB has an additional interception point called `receive_request_service_contexts`, which is invoked by a POA before it dispatches a servant manager. This interception point helps reduce the overhead of making unnecessary upcalls to a servant. For example, in the CORBA Security Service framework this interceptor can be used to inspect security-related credentials piggybacked in a service context list entry. If the credentials are valid the upcall will occur; if not, an exception will be returned to the client.

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 ORB via an IDL interface defined by the Portable



```
PORTABLE INTERCEPTOR API :
1) send request()/send poll()
2) receive request service context ()
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

Interceptor Specification. Irrespective of the type of the interface or the operation that is being 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. Different hook methods will be called at different points in the interceptor chain, however, *e.g.*, 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.

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. Monitoring and debugging utilities can take advantage of this feature to log information associated with each operation invocation. Naturally, care must be taken when implementing nested invocations in an interceptor to act only on targeting objects that it intends to affect and not cause infinite recursion.

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 ORB interface. Interceptors cannot however, modify parameters or return values. This request/reply information is encapsulated in an instance of a class derived from `RequestInfo`, which contains the information listed above, per-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: Although request interceptors cannot change parameters or the return value of an operation, they can manipulate *service contexts* that propagate between the clients and servers. Service contexts are messages that can be piggybacked in operation requests and replies to exchange "out-of-band" information, such as authentication credentials, transaction contexts, operation priorities, or policies associated with requests.

Each service context entry has a unique service context id that applications and CORBA components can use to extract the appropriate service context. For example, the CORBA Security Service uses interceptors to insert user identity via request service contexts. Likewise, the CORBA Transaction Service uses 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.

Location forwarding: Request interceptors can be used to forward a request to a different location that may or may not be known to the ORB *a priori*. This is done via the standard CORBA `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. The replicated IOR can be used as the forward object in this exception. When the object dies for some reason, and this is notified to the interceptor by some polling mechanism, this exception can be raised even before the POA tries to make an upcall. The "permanent" flag can be used to set the replicated IOR as the one to which all future invocations will be made, thereby providing the building blocks to improve application fault tolerance [9, 10].

Multiple interceptors: Multiple interceptors can be registered with the ORB for each interception point. This type of interceptor is handled according to the following rules:

- Only one starting interception point can be called for an invocation;
- Only one ending interception point can be called for an invocation;
- There can be multiple intermediate interception points;
- Intermediate interception points cannot be invoked in case of an exception;
- The ending interception point 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 into the stack after it completes successfully. When the invocation returns, the interceptors are popped off the stack and invoked in reverse order. The flow-stack model ensures that only interceptors that have "seen" the 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_other` operation is invoked on the reply path to the client and is received at the client in the `receive_other` interceptor hook. On asynchronous calls, the reply does not immediately follow the request, so the `receive_other` interceptor hook is called. To cater to specific cases, such as time-independent invocation (TII), the interceptor hook `send_poll` is called by the client ORB. For example, a TII application can poll for a response to a request sent previously by the client, which can be obtained by the `send_poll` interceptor hook.

2.2.2 IOR Interceptors

IIOP version 1.1 introduced an entity called a *component*, which contains a list of *tagged components* in an interoperable object reference (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 dealing with security, server thread priorities, network connections, CORBA policies, or other domain-specific information.

The original IIOP 1.1 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 implementation portability. The Portable Interceptors specification resolves this problem by defining *IOR interceptors*.

IOR interceptors are objects invoked by the ORB when it creates IORs in order to allow the IOR to be customized, *e.g.*, by inserting various tagged components. Request interceptors access operation-related information via `RequestInfo`s, whereas IOR interceptors access IOR-related information via `IORInfo`s. Figure 4 illustrates the lifecycle of IOR interceptors.

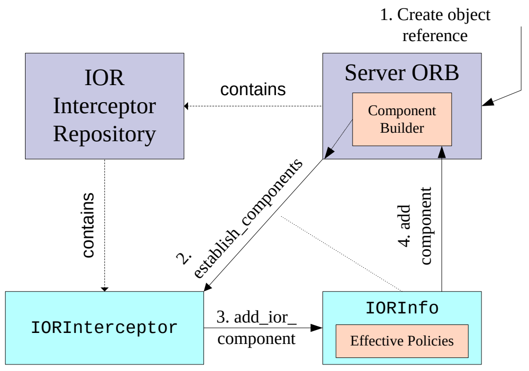


Figure 4: IOR Interceptors

2.3 Evaluating Alternative Meta-Programming Mechanisms for ORB Middleware

We have presented an overview of smart proxies and interceptors above. We now evaluate these two mechanisms, and then compare and contrast them with other meta-programming mechanisms, such as 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 can be used to 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, if an IOR is invalid, an interceptor can discovered this before the POA makes the upcall, which provides an opportunity for redirecting the operation.

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

Invocation points: A smart proxy invocation point occurs whenever an operation is invoked on a stub, whereas interceptors are invoked at many points, including at the IOR creation time, before call is sent by the POA to the servant, etc.

Cardinality: A client proxy can have only a single smart proxy, whereas multiple interceptors can be registered with the ORB and will be invoked in FIFO registration order.

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 interceptors to change operation parameters or return values.

Overhead: A smart proxy mechanism incurs a single extra method call per-operation, whereas interceptors can incur additional overhead to access request information because information related to the request is bundled into `anys`, 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 portable shortly, as soon as the Portable Interceptor specification is ratified as a CORBA standard.

In general, design problems that require pre-invocation extensions are well-suited for smart proxies. Conversely, portable interceptors provide a suitable solution for applications that require a semantically richer, yet somewhat more expensive, meta-programming abstraction.

2.3.2 Servant Managers

The CORBA POA specification [2] 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 two hook methods called `preinvoke` and `postinvoke` 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 the CORBA Component Model to perform various resource management activities before dispatching an operation to a servant.

A servant locator is similar to an interceptor in several respects. For example, 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 access to a limited subset of the request-related information.

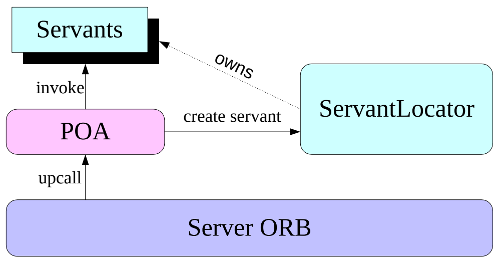


Figure 5: Using a Servant Locator in the CORBA Component Model

As a consequence, they are more tightly coupled with servant implementations than are interceptors.

2.3.3 Pluggable Protocols Frameworks

Another type of meta-programming mechanisms provided by some DOC middleware are *pluggable protocols frameworks* [11, 12]. These frameworks decouple the ORB's communication protocols from its component architecture, thereby allow application developers to add new protocols without requiring application changes.

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

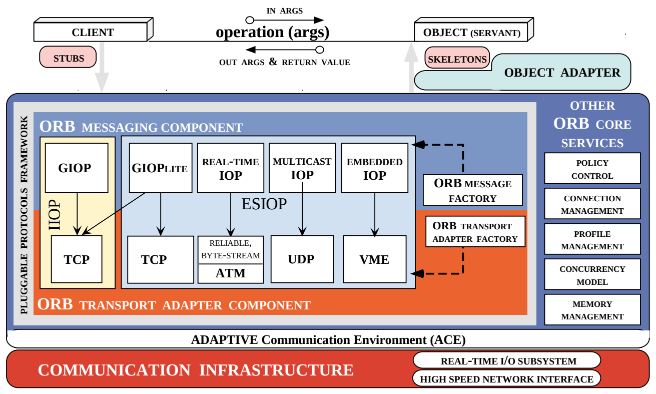


Figure 6: TAO's Pluggable Protocols Framework Architecture

use the Facade pattern [5] to access the mechanisms provided by TAO's pluggable protocols framework. Thus, applications can (re)configure custom protocols without requiring global changes to themselves or the ORB.

As with interceptors and smart proxies, pluggable protocols frameworks are a meta-programming mechanism that adds functionality to ORBs. Whereas other two mechanisms alter the semantic of objects, however, pluggable protocols frameworks alter the ORB message delivery 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 message delivery mechanism at the level of object reference. 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. When

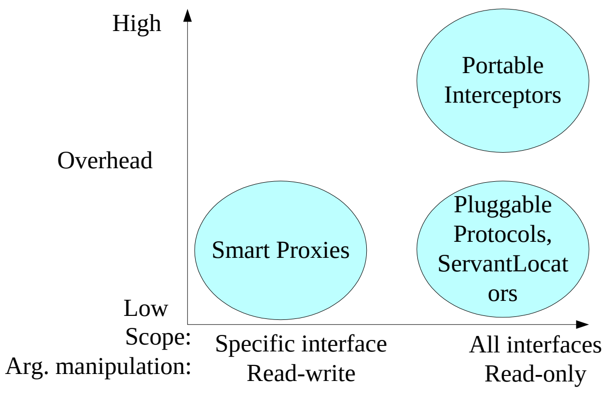


Figure 7: Comparing Alternative Meta-programming Mechanisms

combined with patterns, such as Component Configurator [1] and OS features, such as explicit dynamic linking [13], all these meta-programming mechanisms can be configured dynamically into CORBA clients and servers.

3 Programming with Smart Proxies and Portable Interceptors

Before delving into the design challenges associated with implementing smart proxies and interceptors, this section describes two examples that illustrate how these meta-programming mechanisms can be used to adapt existing systems as requirements change with minimal impact on existing client and server applications. 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

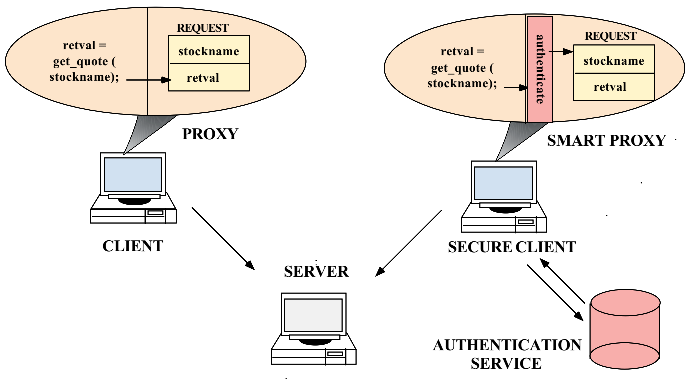


Figure 8: A Secure Transaction System Using Smart Proxies

Figure 8. 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 [14].

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

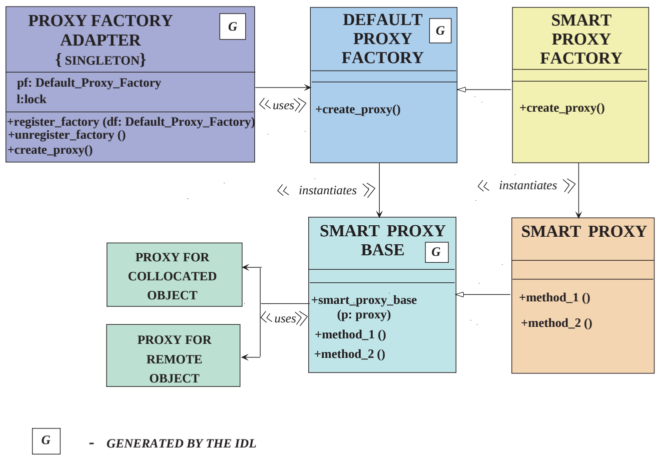


Figure 9: The Classes in TAO's Smart Proxy Framework

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 [5] 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.

• **Default proxy factory,** which is the default factory object that returns the default proxy, *i.e.*, the stub that communicates with remote target objects. This factory registers itself with the adapter singleton during program initialization. It is deleted when it is replaced by another proxy factory or 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 automatically registered with the singleton [5] proxy factory adapter via a constructor in the base class. 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. This design ensures that new object references to target objects of the same interface will use the default proxy unless a new smart proxy factory is installed explicitly for the new object. This new smart proxy factory could either be another instance of the one that was created earlier or a completely 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 [5] 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 Smart_Quoter_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 remote 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 Smart_Stock_Quoter_Proxy :
  public Stock_Quoter_Smart_Proxy_Base
{
  // Smart proxy method.
  CORBA::Long
  Smart_Stock_Quoter_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 that creates smart proxies 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.
  Smart_Quoter_Factory *quoter_factory
    = new Smart_Quoter_Factory;

  // ... 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 allocated on the heap since TAO's smart proxy framework classes assume ownership of 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 quite precisely.

Smart proxies can also be installed dynamically into an application via the Component Configurator pattern [1]. 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 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 accomplish this transparently in CORBA, 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:

```
 // Create and Install the client interceptor.
  PortableInterceptor::
  ClientRequestInterceptor_var
    interceptor = new
      Secure_Client_Request_Interceptor
        (orb.in ());

  orb->_register_client_interceptor
    (interceptor);
```

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
    (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 ...

    // 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` hook method obtains the service context from the service context list stored in the `ServerRequestInfo` object and verifies the password. It also 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:

```
// Create and install the server interceptor.
PortableInterceptor::
ServerRequestInterceptor_var
  interceptor = new
    Secure_Server_Request_Interceptor
      (orb.in ());

orb->_register_server_interceptor
  (interceptor);
```

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. We can add authorization capabilities using the same interceptor by locating it at a different interception point. 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 [5] we applied to resolve the key design challenges we faced during the 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 of the key design challenges we faced while refactoring the 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 little or no changes to client application code.

Problem: TAO's IDL compiler originally 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 overcome at the stub-generation level.

Solution → Apply the Factory Method, Adapter, and Singleton patterns: To configure TAO to create different types of proxies transparently, we applied the Factory Method, Adapter, and Singleton patterns [5] in TAO's IDL-generated code to form a smart proxy framework that provides the necessary flexibility. We used the Factory Method pattern to defer instantiation of various types of meta-objects to subclasses. We used the Adapter pattern to provide a higher level of abstraction for the proxy factories and to delegate creation requests to the appropriate factory. Finally, we used the Singleton pattern to make the proxy factory adapter a singleton that provides 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

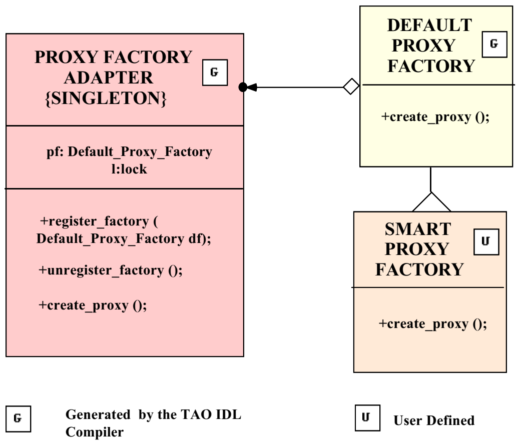

PROXY FACTORY ADAPTER {SINGLETON}
G
pf: Default_Proxy_Factory
l:lock
+register_factory (Default_Proxy_Factory df);
+unregister_factory ();
+create_proxy ();
DEFAULT PROXY FACTORY
G
+create_proxy ();
SMART PROXY FACTORY
U
+create_proxy ();
G
Generated by the TAO IDL Compiler
U
User Defined

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 automatically create the appropriate proxy.

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 [15]. 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 allowing developers 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, *e.g.*, a developer may not want to cache operation results in a collocated smart proxy because calls are already resolved locally. Originally TAO treated the generation of collocated stubs as a special case. It is necessary, however, to distinguish remote and collocated case to take full advantage of this construct. In addition, smart proxies must provide applications with the same interface as default proxies, as well as be able to call down to the default proxy to communicate with remote target objects.

Solution → Apply the Composite pattern: The Composite pattern [5] supports part/whole relationships and allows all objects in such composite structures to be processed uniformly. We applied the Composite pattern to TAO as shown in Figure 11. In this design (1) smart proxy classes inherit from the

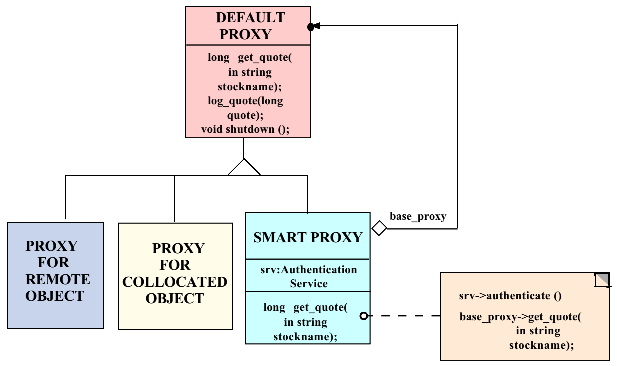

DEFAULT PROXY
long get_quote(in string stockname);
log_quote(long quote);
void shutdown ();
PROXY FOR REMOTE OBJECT
PROXY FOR COLLOCATED OBJECT
SMART PROXY
srv:Authentication Service
long get_quote(in string stockname);
base_proxy
srv->authenticate ()
base_proxy->get_quote(in string stockname);

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

default proxy and (2) also store a pointer to the default proxy to make invocations to remote 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 of the key design challenges faced while refactoring 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, *e.g.*, the result of an operation may be available only after the POA makes an upcall to a servant and the operation executes. Therefore, an ORB must 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 the application. However, this 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 has

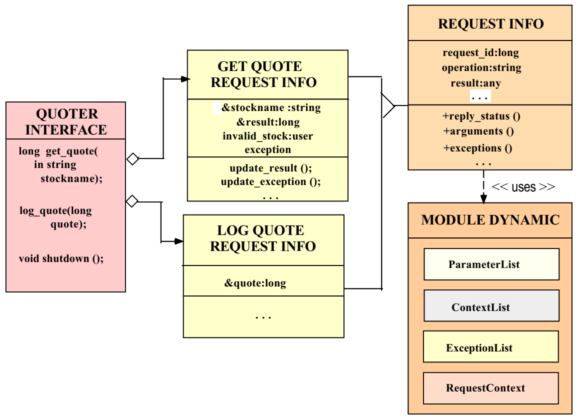


Figure 12: TAO's Portable Interceptor Design

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: Since interceptors can access any request-related information their interface must be sufficiently general to incorporate any type of data. In CORBA, `any` is the generic type that can hold information of any other types, which are stored using type/value tuples. Although `any`s are flexible, they are less efficient and more resource consumptive than other common CORBA data types, such as `long` or `struct`.

In general, not all interceptors installed in an ORB are interested in handling all information nor 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.

Problem: We need to avoid the overhead of `any` insertion operators if installed interceptors are not interested in certain operation information. However, there is no way to predict what interceptors will be interested in *a priori*.

Solution → On-demand creation of operation information: To avoid unnecessary waste of resources, operation information should only be 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 not be created if no interceptor is interested in the information. In TAO, we retrieve this information via types defined in the CORBA `Dynamic` module.

The `Dynamic` module facilitates bundling of request parameters, results, and exceptions into `any` variables that an application interceptor can extract and use. The advantage of using the types provided by the `Dynamic` module in TAO is that they are implemented to minimize the gratuitous waste of storing all information `de facto` into lists of `any`s. 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.

5 Empirical 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 13. The benchmarking programs were compiled using the Sun CC

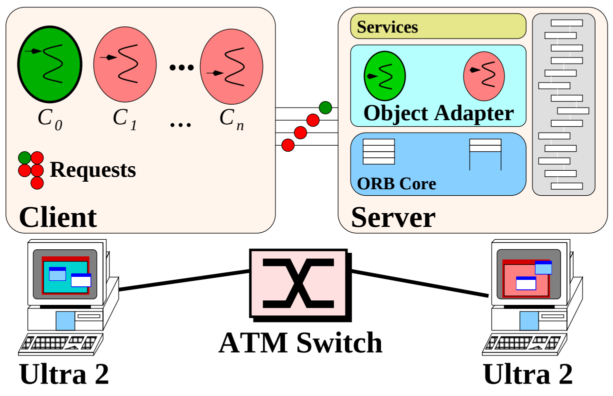


Figure 13: Testbed for Meta-programming Mechanism Benchmarks

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
```

```
{
  // 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 14. This figure illustrates that omitting unnecessary remote operation calls improve the performance by ∼130%, even over a high-speed ATM network.

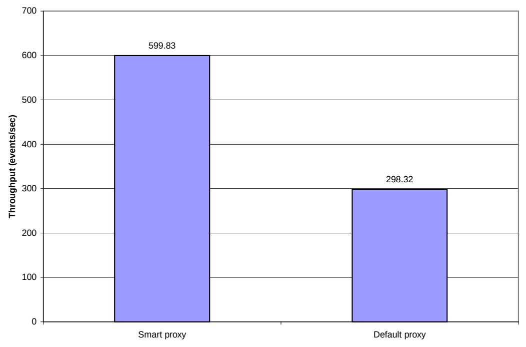


Figure 14: Performance Improvement Using a Smart Proxy to Cache Information

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`, using a program similar to the one shown in Section 3.2. 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 15 shows the cost of supporting and using these various features and configurations in interceptors. In the first configuration (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

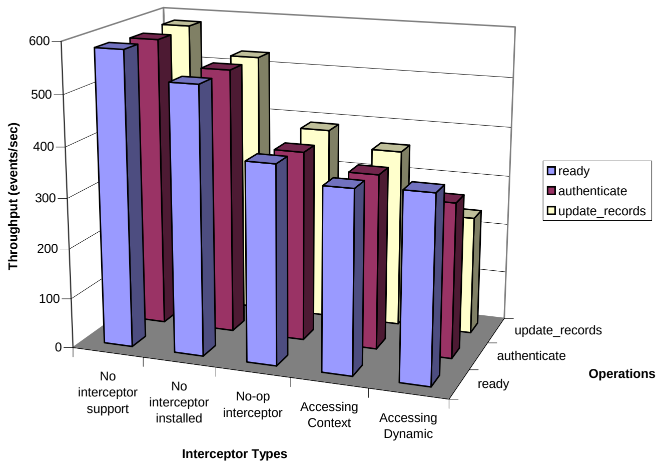


Figure 15: Cost of Using Various Interceptor Features

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 15, 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 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.

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

We also performed the same test using the OMG *Minimum CORBA* configuration [16], 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 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

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

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 [17] by applying Aspect-Oriented Programming (AOP) [18] 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-like mechanism based on the concept of "flexible bindings" [19]. 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 multi-threaded servers [20, 21, 22].

dynamicTAO: The dynamicTAO reflective ORB [23] supports interceptors for monitoring and security. Particular interceptor implementations are loaded into dynamicTAO using the Component Configurator pattern [1]. 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 [24]. 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 [25] describe how to implement interceptors in COM. The concept they use to implement interceptors is similar to TAO's collocated stub [15]. 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. Although DOC middleware shields developers from many distribution challenges, it requires *meta-programming* mechanisms to allow 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.

Smart proxies and interceptors are two examples of meta-programming mechanisms that can both monitor and modify invocation behavior via meta-objects. There are tradeoffs and limitations with each mechanism, *e.g.*:

- Smart proxies perform better, consume less memory, but only apply to specific interfaces accessed by clients. In particular, smart proxies can only influence the behavior at the beginning of an invocation.

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.

- Interceptors are more generic, can be applied to either servers or clients, and can access operation-specific information. Therefore, they provide a more effective meta-programming mechanism to handle advanced features, such as authentication and authorization, transparently end-to-end.

Interceptors also incur more overhead than smart proxies, however, 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.

This paper describes how we have implemented smart proxies and interceptors into TAO, which is an implementation of CORBA that is targeted for applications with high-performance and real-time QoS requirements. 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 [8] that is in the final stages of approval 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] 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.

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

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

[4] 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.

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

[6] 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.

[7] 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.

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

[9] L. Moser, P. Melliar-Smith, and P. Narasimhan, "A Fault Tolerance Framework for CORBA," in *International Symposium on Fault Tolerant Computing*, (Madison, WI), pp. 150–157, June 1999.

[10] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the 2^{nd} International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 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] W. W. Ho and R. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, pp. 375–390, Apr. 1991.

[14] 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.

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

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

[17] 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.

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

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

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

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

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

[23] 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.

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

[25] 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.