

# A Flexible and Extensible Object Middleware CORBA and Beyond

Franz J. Hauck<sup>1</sup>, Rüdiger Kapitza<sup>2</sup>, Hans P. Reiser<sup>1</sup>, Andreas I. Schmied<sup>1</sup>  
{franz.hauck, hans.reiser, andreas.schmied}@uni-ulm.de  
kapitza@informatik.uni-erlangen.de

<sup>1</sup>Distributed Systems Lab, University of Ulm, Germany

<sup>2</sup>Department of Computer Science 4, University of Erlangen-Nürnberg, Germany

**Abstract.** This paper presents a CORBA-compliant middleware architecture that is more flexible and extensible compared to standard CORBA. The portable design of this architecture is easily integrated in any standard CORBA middleware; for this purpose, mainly the handling of object references (IORs) has to be changed. To encapsulate those changes, we introduce the concept of a generic *reference manager* with *portable profile managers*. Profile managers are pluggable and in extreme can be downloaded on demand. To illustrate the use of this approach, we present a profile manager implementation for fragmented objects and another one for bridging CORBA to the Jini world. The first profile manager supports truly distributed objects, which allow seamless integration of partitioning, scalability, fault tolerance, end-to-end quality of service, and many more implementation aspects into a distributed object without losing distribution and location transparency. The second profile manager illustrates how our architecture enables fully transparent access from CORBA applications to services on non-CORBA platforms.

**Keywords:** Software architectures for middleware; extensible and reconfigurable middleware; middleware interoperability; fault tolerance; scalability; CORBA

## 1 Introduction

Middleware systems are heavily used for the implementation of complex distributed applications. Current developments like mobile environments and ubiquitous computing lead to new requirements that future middleware system will have to meet. Examples for such requirements are the support for self-adaptation and self-optimisation as well as scalability, fault-tolerance, and end-to-end quality of service in the context of high dynamics. Heterogeneity in terms of various established middleware platforms calls for cross-platform interoperability. In addition, not all future requirements can be predicted today. A proper middleware design should be well-prepared for such future extensions.

CORBA is a well-known standard providing an architecture for object-based middleware systems [5]. CORBA-based applications are built from distributed objects that can transparently interact with each other, even if they reside on different nodes in a distributed environment. CORBA objects can be implemented in different programming languages. Their interface has to be defined in a single, language-independent interface description language (IDL). Problem-specific extensions allow to add additional features to the underlying base architecture.

This paper discusses existing approaches towards a more flexible middleware infrastructure and proposes a novel modularisation pattern that leads to a flexible and extensible object middleware. Our design separates the handling of remote references from the object request broker (ORB) core and introduces the concept of ORB-independent *portable profile managers*, which are managed by a generic *reference manager*. The profile managers encapsulate all tasks related to reference handling, i.e., reference creation, reference marshalling and unmarshalling, external representation of references as strings, and type casting of representatives of remote objects. The profile managers are independent from a specific ORB, and may even be loaded dynamically into the ORB. Only small modifications to existing CORBA implementations are necessary to support such a design.

Our architecture enables the integration of a fragmented object model into CORBA middleware platforms, which allows transparent support of many implementation aspects of complex distributed systems, like partitioning, scalability, fault-tolerance, and end-to-end quality-of-service guarantees. It also provides a simple mechanism for the integration of cross-platform interoperability, e.g., the integration with services running on non-CORBA middleware platforms, like Jini or .NET remoting. Our design was named *AspectIX* and implemented as an extension to the open-source CORBA implementation *JacORB*, but is easily ported to other systems.

This paper is organised as follows: The following section discusses the monolithic design of most current middleware systems in more detail. It addresses the extension features of CORBA and discusses their lack of flexibility. Section 3 explains our novel approach to middleware extensibility based on a generic reference manager with pluggable profile managers. In Section 4, two CORBA extensions and the corresponding profile managers are presented: One for integrating the powerful fragmented-object model into the system, and one for transparently accessing Jini services from CORBA applications. Section 5 evaluates the implementation effort and run-time overhead of our approach, and Section 6 presents some concluding remarks.

## 2 Middleware Architecture and Extension Points

### 2.1 Standard CORBA Architecture

CORBA uses a monolithic object model: CORBA objects have to reside on a specific node and are transparently accessed by local proxies called stubs. The stubs use an RPC-based communication protocol to contact the actual object, to pass parameters, and to receive results from object invocations. A CORBA-based middleware implementation is free to choose the actual protocol, but has to support the Internet Inter-ORB Protocol (IIOP) for interoperability.

CORBA uses *interoperable object references* (IORs) to address remote objects. The IOR is a data structure composed of a set of *profiles*. According to the standard, each profile may specify contact information of the remote object for one specific interaction protocol; for interoperability between ORBs of different vendors, an IIOP profile needs to be present. In addition to protocol profiles, the IOR may contain a set of *tagged components*. Each tagged component is a name-value pair with a unique tag registered with the OMG and arbitrary associated data; these components define protocol-independent information, like a unique object ID.

In standard CORBA, IORs are created internally at the server ORB. A server application creates a servant instance, registers the servant with the ORB (or, to be more specific, with an object adapter of the ORB). Usually this IOR contains an automatically created IOP profile that contains hostname, port, object adapter name, and object id for accessing the object via IOP. Additionally it may contain other profiles representing alternative ways to access the object.

An IOR can be passed to remote clients, either implicitly or explicitly. If a reference to a remote object is passed as a parameter or return value, the IOR data structure will be implicitly serialised and transferred. Upon deserialisation, the receiving client ORB automatically instantiates a local stub for accessing the remote object, initialised with the information available in the IOR. If multiple profiles exist in the IOR, the ORB may use a vendor-specific strategy to select a single profile that is understood by the ORB. The IOP profile should be understood by all ORBs. Beside implicit transfer, an explicit transfer is possible. The server application may call a `object_to_string` method at the ORB, which serialises the IOR and transforms it to a string, an IOR-URL. This string can be transferred to a client; the client application may call the local ORB method `string_to_object` to create the stub for the remote object referenced by the IOR.

## 2.2 Status Quo of Extensible Middleware

Many practical tasks—e.g., authorisation, security, load balancing, fault tolerance, or special communication protocols—require extensions to this basic model. For example, fault-tolerant replication requires that multiple communication addresses (i.e., of all replicas) are known to a client. Typically this means that these addresses have to be encoded into the remote reference; binding to and invoking methods at such a remote object require a more complex handling at the client side compared to the simple stub-service design. As a second example, a peer-to-peer-like interaction between users of a service might sometimes be desired. In this case, the client-server structure needs to be completely abandoned.

A good example for the lack of extensibility of standard CORBA is the fault-tolerant CORBA standard (FT-CORBA) [5, Chapter 23]. It was not possible to define this standard in a way that all platform implementations are portable across different ORBs. Instead, each FT-CORBA-compliant middleware has its vendor-specific implementation inside of a single ORB. A system design such as we envision allows to implement a generic “FT-CORBA plugin” that makes any ORB, independent of its vendor, aware of fault-tolerance mechanisms.

Existing concepts for interception, custom object adapters, and smart proxies provide some mechanisms for such extensions. In part, they are included in the CORBA standard; in part, they are only available in non-standardised middleware implementations.

The portable interceptor specification supports interception within the official CORBA standard. The specification defines request interceptors and IOR interceptors as standardised way to extend the middleware functionality. Using request interceptors, several hooks may be inserted both at client and at server side to intercept remote method calls. These hooks allow to redirect the call (e.g., for load balancing), abort it with an exception (e.g., for access restriction), extract and modify context information embedded in the request, and perform monitoring tasks. A direct manipulation of the request is not permitted by the specification. Multiple request interceptors may be used simultaneously. Interceptors add

additional overhead on each remote method invocation; furthermore, they do not allow to modify the remote invocations completely. IOR interceptors, on the other hand, are called when a POA needs to create an IOR for a service. This allows to insert additional data into the IOR (e.g., a tagged component for context information that is later used in a request interceptor).

CORBA allows to define custom POA implementations. The POA is responsible to forward incoming invocation requests to a servant implementation. This extension point allows to integrate server-side mechanisms like access control, persistence, and life-cycle management. It has, however, no influence on the interaction of clients with a remote service.

Beside the standardised IIOP protocol, any CORBA implementation may support custom invocation protocols. Additional IOR profiles may be used for this purpose. However, establishing such extensions is not standardised and every vendor may include his own proprietary variant, which limits the interoperability of this approach.

Smart proxies are a concepts for extending ORB flexibility, which is not yet standardised by the OMG, but which is implemented by some ORB vendors, e.g., in the *ACE ORB/Tao* [11]. They allow to replace the default CORBA stub by a custom proxy that may implement some extended functionality. As such, they allow to implement parts of the object's functionality at the client side.

Closely related to our research is the work on *OpenORB* at Lancaster University [1]. This reflective middleware project uses reflection to define dynamic (re)configuration of componentised middleware services. One main difference to our design is that it completely restructures the middleware architecture, whereas our concept with a reference manager and pluggable profile managers is integrated in any existing CORBA implementation with only minimal modifications. It nevertheless provides equal flexibility in reconfigurations. Component technology could be used in the internal design of complex profile managers.

*PolyORB* [10] is a generic middleware system that aims at providing a uniform solution to build distributed applications. It supports several *personalities* both at the application-programming interface (API) level and the network-protocol level. The personalities are compliant to several existing standards. This way, it provides middleware-to-middleware interoperability. Implementations of personalities are specific to PolyORB, unlike our profile managers, which are intended to be portable between different ORB implementations. We do not address the issue of genericity at the API level.

### 3 Designing a Generic Reference Manager

The fundamental extension point of an object middleware is the central handling of remote references. It is the task of any object middleware to provide mechanisms to create remote references, to pass them across host boundaries, and to use them for remote invocations. As explained above, merely providing extension points at the invocation level is insufficient for several complex tasks. The essential point of our work is to provide a very early extension point by completely separating the reference handling from the middleware core.

The impact of such a design is not only that a single middleware implementation gets more flexible. It is also highly desirable to provide this extensibility in an vendor-independent way. That is, an extension module should be portable across different middleware im-

plementations. Furthermore, these extensions should be dynamically pluggable and, in the extreme, be loaded on demand by the middleware ORB.

Our design provides such a middleware architecture. It is currently designed as an extension to standard CORBA, and maintains interoperability with any legacy CORBA system. Our design represents, however, a generic design pattern that easily applies to any other object middleware.

The only prerequisite made is that remote references are represented by a sufficiently extensible data structure. In CORBA, the Interoperable Object Reference (IOR) provides such a data structure. Each profile of the IOR represents an alternative way to contact the object. Each profile type has its own data-type definition, described in CORBA IDL. Hence, at the IOR level, CORBA is open to arbitrary extensions. The IOR handling, however, is typically encapsulated in the internals of a CORBA-compliant ORB implementation. Currently, if a vendor uses the power of IORs for custom extensions, these will only be implemented internally in the respective ORB. The extension will not be portable.

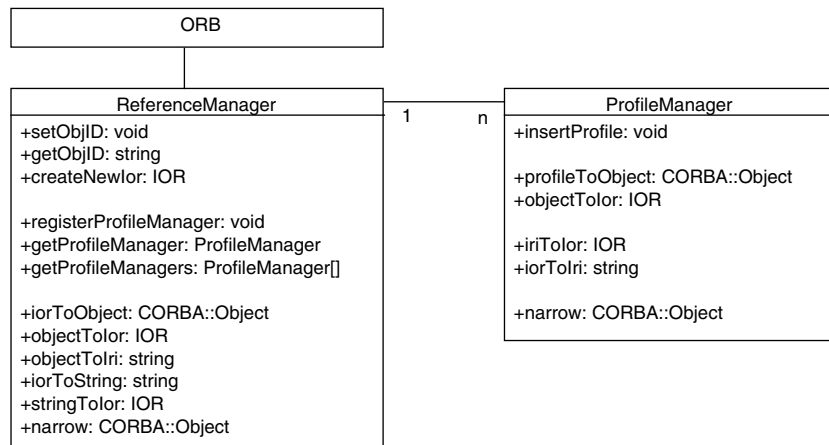
### 3.1 Overview of our Design

Our approach introduces a generic interface for plugging in portable extension modules for all tasks related to IOR profile handling. This makes it easy to support extended features like fault-tolerant replication, a fragmented object model, or transparent interaction with other middleware systems. This improves the flexibility of a CORBA middleware. We factored out the IOR handling of the ORB and put it into pluggable modules. This way, custom handlers for IOR profiles may be added to the ORB without modifying the ORB itself. Dynamically downloading and installing such handlers at run-time further contributes to the richness of this approach.

Factoring out the basic remote-reference handling of the ORB core into a pluggable module affects five core functions of the middleware: first, the creation of new object references; second, the marshalling process, which converts object references into an externally meaningful representation (i.e., a serialised IOR); third, the unmarshalling process, which has to convert such representation into a local representation (e.g., a stub, a smart proxy, or a fragment); and fourth, the explicit binding operation, which turns some symbolic reference (e.g., a stringified IOR) into a local representation and vice versa. A fifth function is not as obvious as the others: The type of a remote object reference can be changed as long as the remote object supports the new type. In CORBA this is realised by a special *narrow* operation. As in some cases the narrow operation needs to create a new local representation for a remote object, this operation has to be considered too.

An extension to CORBA will have to change all five functions for its specific needs. Thus, we collect those function in a module that we call a *profile manager*. A profile manager is usually responsible for a single type of IOR profile, but there may be reasons to allow profile managers to manage multiple profile types. Profile managers are pluggable

modules. A part of the ORB called *reference manager* manages all available profile managers and allows for registration of new profile-manager modules.



**Fig. 1 UML Class Diagram of the CORBA Extension**

The basic design can be found as a UML class diagram in Fig. 1. Sometimes an application needs to access the reference manager directly. It can retrieve a reference to the reference manager pseudo object from the ORB by calling `resolve_initial_references()`, a generic operation for resolving references to system-dependent objects. At the reference manager, profile managers can be registered. As profile managers are responsible for a single or for multiple IOR profile types, the registration requires a parameter identifying those profile types. For identification a unique profile tag is used. Those tags are registered with the OMG to ensure their uniqueness. With the registration at the reference manager, it is exactly known which profile managers can handle what profile types. Several tasks of reference handling are invoked at the reference manager and forwarded to the appropriate profile manager. The architecture resembles the chain-of-responsibility pattern introduced by Gamma et al. [2].

### 3.2 Refactoring the Handling of References

In the following, we describe the handling of the five core functions of reference handling in our architecture:

**Creating object references.** In traditional CORBA, new object references are created by registering a servant at a POA of the server. The POA usually maintains a socket for accepting incoming invocation requests, e.g., in form of IIOP messages. The POA encodes the contact address of the socket into an IIOP profile and creates an appropriate IOR. The details of the POA implementation varies from ORB to ORB. In general, the registration at the POA creates an IOR and an internal data structure containing all necessary information for invocation handling.

To be as flexible as possible our extension completely separates the creation of the IOR from the handling in a POA. The creation of an IOR requires the invocation of `create-`

`NewIor()` at the reference manager. As standard CORBA is not able to clearly identify object references referring to the same object, we added some operations to the reference manager that allow for integrating a universally unique identifier (UUID) into the IOR. The UUID is stored as a tagged component and in principle can be used by any profile manager.

For filling the IOR with profiles, an appropriate profile manager must be identified. Operations at the reference manager allow the retrieval of profile managers being able to handle a specific profile type. A profile manager has to provide an operation `insertProfile()` that adds a profile of a specific type into a given IOR. This operation has manager-dependent parameters so that each manager is able to create its specific profile. Instead of creating the IOR itself, the POA of our extended ORB has to create the IOR by asking the IIOP profile manager for adding the appropriate information (host, port, POA name and object ID) to a newly created IOR. As an object may be accessible by multiple profiles, an object adapter can ask multiple profile managers for inserting profiles into an IOR.

**Marshalling of object references.** A CORBA object passed as a method parameter in a remote invocation needs to be serialised as an IOR. In classic CORBA, this is done „magically“ by the ORB by accessing internal data structures. In the Java language mapping, for example, a CORBA object reference is represented as a stub that delegates to a sub-type of `org.omg.CORBA.Delegate`. Instances of this type store the IOR.

In our architecture, we cannot assume any specific implementation of an object reference as each profile manager may need a different one. Thus, there is no generic way to retrieve the IOR to be serialised. Instead, the reference manager is asked to convert the object reference into an IOR that in the end can be serialised. Therefore, the reference manager provides an operation called `objectToIor()`. The reference manager, in turn, will ask all known profile managers to do the job. Profile managers will usually check whether the object reference is an implementation of their middleware extension. If so, the manager will know how to retrieve the IOR. If not, the profile manager will return a null reference, and the reference manager will turn to the next profile manager.

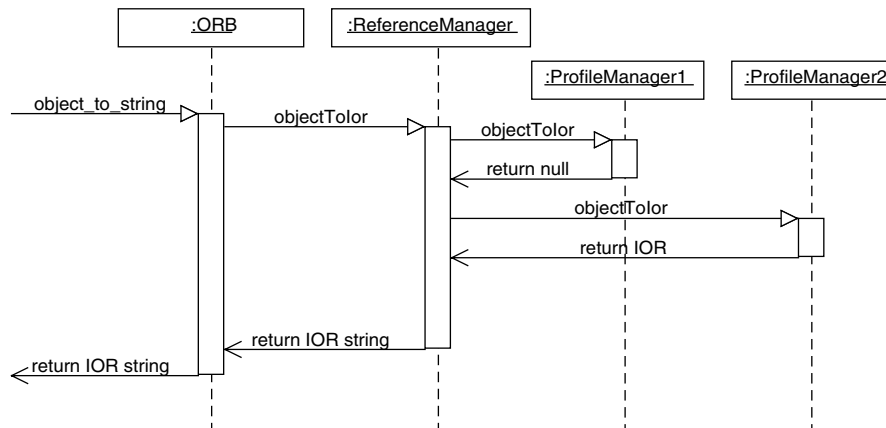
**Unmarshalling of object references.** If a standard ORB receives a serialised IOR as a method parameter or return result, it implicitly converts it into a local representation and passes this representation (typically a stub) to the application. This creation of the stub needs to be factored out of the marshalling system of the ORB to handle arbitrary reference types.

Our design delegates the object creation within unmarshalling to the reference manager by calling `iorToObject()`. The reference manager maintains an ordered list of profile types and corresponding profile managers. According to this list, each profile manager is asked to convert the IOR in a local representation by calling `profileToObject()`. This way the reference manager already analyses the contents of the IOR and only asks those managers that are likely to be able to convert the IOR into an object reference. A profile manager checks the profile and the tagged components, and tries to create a local representation of the object reference. For example, an IIOP profile manager will analyse the IIOP profile. A CORBA-compliant stub is created and initialised with the IOR. The stub is returned to the reference manager, which returns it to the application. If a profile manager is not able to convert the profile or not able to contact the object for arbitrary reasons, it will throw an exception. In this case the reference manager will follow its list and ask the next profile manager. If none of the managers can deal with the IOR, an exception is thrown

to the caller. This is compatible with standard CORBA for the case that no profile is understood by the ORB.

The order of profile types and managers defines the ORB-dependent strategy of referencing objects. As the first matching profile type and manager wins, generic managers (e.g., for IIOP) should be at the end of the list whereas more specific managers should be at the beginning.

**Explicit binding to remote references.** A user application may explicitly call the ORB method `string_to_object`, passing some kind of stringified representation of the references. Usually, this may either be a string representation of the marshalled IOR, or a *corbaloc* or *corbaname* URI.



**Fig. 2** Sequence Chart for `ORB::object_to_string`

This ORB operation can be split into two steps: First, the string is parsed and converted into an IOR object. As the stringified IOR can have an extension-specific IRI format—an IRI is the internationalised version of a URI—each profile manager is asked for conversion by using `iriToIor()`. The generic IOR format will be handled by the reference manager itself. Second, this IOR object is converted into a local representation using the same process as used with unmarshalling.

Calling `object_to_string()` is handled in a similar way. First, the reference passed as parameter is converted into an IOR object in the same way as for marshalling. Second, the IOR object is converted into a string. The IOR is encoded as a hex string representing an URI in the IOR schema. The complete interaction is shown as sequence chart in Fig. 2.

As sometimes an application may want to convert an object reference into a more-readable IRI, our extension also provides an operation called `objectToIri()`. In a first step, the object reference is once again converted into an IOR. The second step, the conversion into an IRI, is done with the help of the profile managers by invoking `iorToIri()`. Those may provide profile-specific URL schemes that may not be compatible to standard

CORBA. As the conversion of an IRI into an object reference can be handled in the profile manager this is not a problem.

**Narrow operation.** The narrow operation is difficult to implement. In the Java language mapping the operation is located in a helper class of the appropriate type, expecting an object reference of arbitrary other type. The implementation in a standard ORB assumes an instance compatible to the basic stub class, which knows a delegate to handle the actual invocations. After successfully checking the type conformance, the helper class will create a new stub instance of the appropriate type and connect it to the same delegate. With any CORBA extension it cannot be assumed that object references conform to the basic stub class.

In our extension the helper class invokes the operation `narrow()` at the reference manager. Beside the existing object reference a qualified type name of the new type is passed to the operation as a string. The reference manager once again will call every profile manager for the narrow operation. A profile manager can check whether the object reference belongs to its CORBA extension. If yes, the manager will take care of the narrow operation. If not, a null result is returned and the reference manager will turn to the next profile manager.

As a profile manager has to create a type-specific instance, it can use the passed type name to create that instance. In languages that provide a reflection API (e.g., Java) this is not difficult to realise. In other languages a generic implementation in a profile manager may be impossible. Another drawback is that reflection is not very efficient. An alternative implementation is the placement of profile-specific code into the helper class (or in other classes and functions of other language mappings). Our own IDL compiler called IDLflex [8] can easily be adapted to generate slightly different helper classes. As a compromise helper classes may have profile-specific code for the most likely profiles, but if other profiles are used, the above mentioned control flow through reference and profile managers is used. Thus, most object references can be narrowed very fast and the ORB is still open to object-reference implementations of profile managers that may have even be downloaded on demand.

## 4 Extensions to CORBA Based on Profile Managers

This section illustrates two applications of our design. The first examples presents the AspectIX profile manager, which integrates fragmented object into a CORBA middleware. The second examples provides a transparent gateway from CORBA to Jini.

### 4.1 AspectIX Profile Manager

The AspectIX middleware supports a fragmented object model [4,7]. Unlike the traditional RPC-based client-server model, the fragmented object model does no longer distinguish between client stubs and the server object. From an abstract point of view, a fragmented object is an entity with unique identity, interface, behaviour, and state, as in classic object-oriented design. The implementation, however, is not bound to a certain location, but may be distributed arbitrarily over various fragments. Any client that wants to access the fragmented object needs a local fragment, which provides an interface identical to that of a tra-

ditional stub. This local fragment may be specific for this object and this client. Two objects with the same interface may lead to completely different local fragments.

This internal structure gives a high degree of freedom on where state and functionality of the object is located and how the interaction between fragments is done. The internal distribution and interaction is not only transparent on the external interface, but may even change dynamically at run-time. Fragmented objects can easily simulate the traditional client-server structure by using the same fragment type at all client locations that works as a simple stub. Similarly, the fragmented object model allows a simple implementation of smart proxies by using the smart proxy as fragment type for all clients. Moreover, this object model allows arbitrary internal configurations that partition the object, migrate it dynamically, or replicate it for fault-tolerance reasons. Finally, the communication between fragments may be arbitrarily adjusted, e.g., to ensure quality-of-service properties or use available special-purpose communication mechanisms. All of these mechanisms are fully encapsulated in the fragmented objects and are not directly visible on the outer interface that all client application use.

Supporting a fragmented object model is clearly an extension to the CORBA object model. With our architecture it is very easy to integrate the new model. Just a new profile manager has to be developed and plugged into our ORB. When a client binds to a fragmented object, a more complex task than simply loading a local stub is needed. In our system, the local fragment is internally composed of three components: the *View*, the *Fragment Interface (FIfc)*, and the *Fragment Implementation (FImpl)*. The fragment implementation is the actual code that provides the fragment behaviour. The fragment interface is the interface that the client uses. Due to type casts, a client may have more than one interface instance for the same fragmented object. Interfaces are instantiated in CORBA `narrow` operations; all existing interfaces delegate invocations to the same implementation. The *View* is responsible for all management tasks; it stores the object ID and IOR, keeps track of all existing interfaces, and manages dynamical reconfigurations that exchange the local *FImpl*. The management needs to update all references from *FIfc*s to the *FImpl* and has to coordinate method invocations at the object that run concurrently to reconfigurations.

To integrate such a model into a profile-manager-aware CORBA system, it is necessary to create IORs with a special profile for fragmented objects (APX profile), and to instantiate the local fragment (*View-FImpl-FIfc*) when a client implicitly or explicitly binds to such an IOR.

The IOR creation is highly application specific, thus it is not fully automatic as in traditional CORBA. Instead, the developer of the fragmented object may explicitly define, which information needs to be present in the object. The reference manager creates an empty IOR for a specified IDL type, and subsequently the APX profile manager can be used to add a APX profile to this IOR. This profile usually consists of information about the initial *FImpl* type that a client needs to load and contact information on how to communicate with other fragments of the fragmented object. The initial *FImpl* type may be specified as a simple Java class name in a Java-only environment, or as a DLS name (dynamic loading service, [3]) in a heterogeneous environment, to dynamically load the object-specific local fragment implementation. The contact information may, e.g., indicate a unique ID, which is used to retrieve contact addresses from a location service) or a multicast address for a fragmented object that uses network multicast for internal communication.

When an ORB binds to an IOR with an APX profile, the corresponding profile manager first checks, if a fragment of the specific object already exists; if so, a reference to the existing local fragment is returned. Otherwise, a new default view is created and connected to a newly instantiated *FImpl*. Profile informations specifies how this *FImpl* is loaded (direct Java class name for Java-only environments, a code factory reference, or a unique ID for lookup to the global dynamic loading service (DLS)). Finally, a default interface (of type APXObject) is built and returned to the client application.

## 4.2 Jini Profile Manager

Jini is a Java-based open software architecture for network-centric solutions that are highly adaptive to change [9]. It extends the Java programming model with support for code mobility in the networks; leasing techniques enables self-healing and self-configuration of the network. The Jini architecture defines a way for clients and servers to find each other on the network. Service providers supply clients with portable Java objects that implement the remote access to the service. This interaction can use any kind of technology such as Java RMI, SOAP, or CORBA.

The goal of this CORBA extension is to seamlessly integrate Jini services into CORBA. Jini services should be accessible to CORBA clients like any other CORBA object. Jini services offer a Java interface. This interface can be converted into an IDL interface using the Java-to-IDL mapping from the OMG [6]. Our CORBA extension provides CORBA-compatible representatives, special proxy objects that appear as CORBA object references, but forward invocations to a Jini service. Such references to Jini services can be registered in a CORBA naming service and can be passed as parameters to any other CORBA object. CORBA clients do not have to know that those references refer to Jini services.

The reference to a Jini service is represented as a CORBA IOR. The Jini profile manager offers operations to create a special Jini profile that refers to a Jini service. It provides operations to marshal references to such services by retrieving the original IOR from the specialised proxy, to unmarshal IORs to a newly created proxy, and to type cast a proxy to another IDL type.

The Jini profile stores a Jini service ID, and optionally a group name and the network address of a Jini lookup service. The profile manager uses automatic multicast-based discovery to find a set of lookup services where Jini services usually have to register their proxy objects. Those lookup services are asked for the proxy of the service identified by the unique service ID. If an address of a lookup service is given in the profile, the profile manager will only ask this service for a proxy. The retrieved proxy is encapsulated in a wrapper object that on the outside looks like a CORBA object reference. Inside, it maps parameters from their IDL types to the corresponding Java types and forwards the invocation to the original Jini proxy.

Jini services may provide a lease for service usage. In the IOR, a method can be named that is supposed to retrieve a lease for the service. This lease will be locally managed by the profile manager and automatically extended if it is due to expire.

This extension shows that it is possible to encapsulate access to other middleware platforms inside of profile managers. In case of the Jini profile manager our implementation is rather simple. So, return parameters referring to other Jini services are not (yet) converted to CORBA object references. We also did not yet implement Jini IOR profiles that encapsulate

sulate abstract queries: In this case not a specific service ID is stored in the profile, but query parameters for the lookup at the lookup services. This way, it will be possible to create IORs with abstract meaning, e.g., encapsulating a reference to the nearest colour printer service (assumed that such services are registered as Jini services at a lookup service).

## 5 Evaluation

Two aspects need to be discussed to evaluate our design: First, the effort that is needed to integrate our concept into an existing ORB; second, the run-time overhead that this approach introduces. As we used JacORB as basis for our implementation, we compare our implementation with the standard JacORB middleware.

### 5.1 Implementation Cost

The integration of a generic reference manager into JacORB version 2.2 affected two classes: `org.jacorb.orb.ORB` and `org.jacorb.orb.CDROutputStream`. In ORB, the methods `object_to_string`, `string_to_object`, and `_getObject` (which is used for demarshalling) need to be replaced. In addition, the reference manager is automatically loaded at ORB initialisation and made available as initial reference. In `CDROutputStream`, the method `write_Object` needs to be re-implemented to access the reference manager. These changes amount to less than 100 lines of code (LOC). The generic reference manager consists of about 500 LOC; the IIOP profile manager contains 150 LOC in addition to the IIOP implementation reused from JacORB.

These figures show that our design easily integrates into an existing CORBA ORB. Moreover, the generic reference manager and profile managers may be implemented fully independent of ORB internals, making them portable across ORBs of different vendors—with the obvious restrictions to the same implementation programming language.

### 5.2 Run-time Measurements of our Implementation

We performed two experiments to evaluate the run-time cost of our approach; all tests were done on Intel PC 2.66 GHz with Linux 2.4.27 operating system and Java 1.5.0, connected with a 100 MBit/s LAN. In all cases, the generic reference manager in our ORB was connected with three profile managers (IIOP, APX, Jini)

The first experiment examines the binding cost. For this purpose, a stringified IOR of a simple CORBA servant with one IIOP profile is generated. Then, this reference is repeatedly passed to the ORB method `string_to_object`, which parses the IOR and loads a local client stub. Table 1 shows the average time per invocation of 100,000 iterations.

Table 1: Execution time of `ORB::string_to_object` invocations

Standard JacORB 2.2.1	0.22 ms
AspectIX ORB with reference manager	0.28 ms
Overhead	0.06 ms (27%)

The second test analyses the marshalling cost of remote references. An empty remote method with one reference parameter is invoked 100,000 times. These operation involve first a serialisation of the reference at client-side and afterwards a deserialisation and binding at the servant side; all operations are delegated to the reference manager in our ORB. Table 2 shows the results of this test.

Table 2: Complete remote invocation time with one reference parameter

Standard JacORB 2.2.1	0.64 ms
Reference Manager	0.72 ms
Overhead	0.08 ms (12.5%)

Both experiments show, that the increase in flexibility and extensibility is paid for with a slight decrease in performance. It is to be noted that our reference implementation has not yet been optimised for performance, so further improvement might be possible.

## 6 Conclusion

We have presented a novel, CORBA-compliant middleware architecture that is more flexible and extensible than standard CORBA. It defines a portable reference manager that uses dynamically loadable profile managers for different protocol profiles.

The concept is more flexible than traditional approaches. Unlike smart proxies, it does not only modify the client-side behaviour, but allows to modify the complete system structure. In contrast to vendor-specific transport protocols, it provides a general extension to CORBA that allows to implement portable profile managers, which in the extreme may even be dynamically loaded as plug-in at run-time. Different from CORBA portable interceptors, it gives the service developer full control over the IOR creation process, has less overhead than interceptors, and allows arbitrary modification of client requests.

The concept itself is not limit to CORBA. In fact, it defines a generic design pattern for any existing and future middleware platforms. Extracting all tasks related to the handling of remote references into an extensible module allows to create middleware platforms that are easily extended to meet even unanticipated future requirements. Modern developments like ubiquitous computing, increased scalability and reliability demands and so on make it likely that such extensions will be demanded. Our design principle allows to implement future systems with best efficiency and least implementation effort.

We have presented in some detail two applications of our architecture. Besides traditional IIOP for compliance with CORBA, we have implemented a profile manager for fragmented objects and one for accessing Jini services transparently as CORBA objects. Currently, we are working on profile managers to handle fault-tolerant CORBA and a bridge to Java RMI.

## 7 References

- [1] G. Coulson, G. Blar, M. Clarke, N. Parlavantzas: *The design of a configurable and reconfigurable middleware platform*. Distributed Computing 15(2): 2002, pp 109-126
- [2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design patterns*. Elements of reusable object-oriented software. Addison-Wesley, 1995.
- [3] R. Kapitza, F. Hauck: *DLS: a CORBA service for dynamic loading of code*. Proc. of the OTM'03 Conferences; Springer, LNCS 2888, 2003
- [4] M. Makpangou, Y. Gourhand, K.-P. Le Narzul, M. Shapiro: *Fragmented objects for distributed abstractions*. Readings in Distr. Computing Systems, IEEE Comp. Society Press, 1994, pp. 170-186
- [5] Object Management Group: *Common object request broker architecture: core specification, version 3.0.3*; OMG specification formal/04-03-12, 2004
- [6] Object Management Group: *Java language mapping to OMG IDL, version 1.3*; OMG specification formal/2003-09-04, 2003
- [7] H. Reiser, F. Hauck, R. Kapitza, A. Schmied: *Integrating fragmented objects into a CORBA environment*. Proc. of the Net.ObjectDays, 2003
- [8] H. Reiser, M. Steckermeier, F. Hauck: *IDLflex: A flexible and generic compiler for CORBA IDL*. Proc. of the Net.Object Days, Erfurt, 2001, pp 151-160
- [9] Sun microsystems: *Jini technology architectural overview*. White paper, Jan 1999
- [10] T. Vergnaud, J. Hugues, L. Pautet, F. Kordon: *PolyORB: a schizophrenic middleware to build versatile reliable distributed applications*. Proc. of the 9th Int. Conf. on Reliable Software Technologies Ada-Europe 2004 (RST'04),; Springer, LNCS 3063, 2004, pp 106-119
- [11] N. Wang, K. Parameswaran, D. Schmidt: *The design and performance of meta-programming mechanisms for object request broker middleware*. Proc. of the 6th USENIX Conference on Object-Oriented Technology and Systems (COOTS'01), 2001