The following paper was originally published in the
Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems
Portland, Oregon, June 1997

Making CORBA Objects Persistent: the Object Database Adapter Approach

Francisco C. R. Reverbel
Departamento de Ciencia da Computacao, Universidade de Sao Paulo - Brazil
Arthur B. Maccabe
Department of Computer Sciences, University of New Mexico

Making CORBA Objects Persistent: the Object Database Adapter Approach*

Francisco C. R. Reverbel†

reverbel@ime.usp.br

Departamento de Ciência da Computação
Universidade de São Paulo
05508-900 – São Paulo, SP – Brazil

Arthur B. Maccabe

maccabe@cs.unm.edu

Department of Computer Science
University of New Mexico
Albuquerque, NM 87505 – USA

Abstract

This paper discusses a realization of object persistence in a CORBA-based distributed system. In our approach, persistence of CORBA objects is accomplished by the integration of the ORB with an ODBMS. This approach is not limited to pure object-oriented database systems, as the ODBMS may be a combination of a relational DBMS and an object-relational mapper. The design and implementation of an Object Database Adapter that integrates an ORB and an ODBMS with C++ bindings is presented. The ODA uses delegation (rather than inheritance) to connect user-provided implementation classes and IDL-generated classes. Only the user-defined parts of CORBA objects are actually stored in a database. Their IDL-generated parts are dynamically instantiated, in transient memory, by the ODA. Persistent relationships between CORBA objects within a server are not realized at the CORBA level, but at the level of implementation objects. Database traversals and queries can therefore be executed at ODBMS speeds. The paper discusses in some detail a number of implementation issues, such as caching. ODA support to local transactions, ODA interfaces, and CORBA server organization are also examined.

1 Introduction

In spite of its remarkable successes in promoting standards for distributed object systems [14], the Object Management Group (OMG) has not yet settled the issue of object persistence in the Object Request Broker (ORB) environment. The Common Object Request Broker Architecture (CORBA) specification [7] briefly mentions an Object-Oriented Database Adapter that makes objects stored in an object-oriented database accessible through the ORB. This idea is pursued in the Appendix B of the ODMG standard [2], which identifies a number of issues involved in using an Object Database Management System (ODBMS) in a CORBA environment, and proposes an Object Database Adapter (ODA) to realize the integration of the ORB with the ODBMS.

Possibly because this proposal was perceived by many as biased towards object-oriented databases, and hence distant from the mainstream database world, no further OMG specifications have contemplated the ODA approach. Instead, a Persistence Object Service (POS), designed to accommodate the widest possible variety of data stores, was introduced in [8]. So far POS failed to deliver its promise. In response to this fact, the OMG recently issued a request for proposals for POS version 2.0:

> "While the industry posses many products from OMG members that could be considered to be in this space, it is clear that virtually none have compliant POS implementations in their product roadmaps. Most have taken the route of point integrations with ORB products." ([11], page 20)

Meanwhile, recognition that the ODA approach is not exclusive to object-oriented databases seems to have grown in the industry. Object-relational mappers — systems that map C++ classes/objects into relational tables/tuples — have been employed to make relational databases appear as object-oriented ones. Because such mappers implement an ODBMS interface on top of a relational system, they extend to relational databases the applicability of the ODA approach.

The benefits of integrating ORBs and ODBMSs include:

Database Heterogeneity. ORB/ODBMS integration allows the construction of distributed object databases that can be heterogeneous even with respect to the DBMS software running on the database server nodes.

*This research was performed at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 97545, as part of the Sunrise Project.

†Partly supported by a fellowship from the National Research Council of Brazil (CNPq).

"IDL views". Access to database objects through IDL interfaces does not require knowledge of the database schema: changes in the schema are transparent to IDL clients. Interfaces can be defined to expose only data items that certain users are permitted to read or update. IDL interfaces to database objects can therefore play a role analogous to relational views, both for data independence and for authorization purposes.

Language Heterogeneity. Databases can be accessed by CORBA clients written in any language for which a mapping from IDL is defined.

Security. The ORB's remote method invocation mechanism requires much less trust in the client than the data-shipping approach employed by pure object-oriented DBMSs.

This paper discusses the design and implementation of an ODA that integrates an ORB and an ODBMS with C++ bindings. For our purposes, an ODBMS is a system with programming interfaces similar to the ones specified in [2]: it may be a pure object-oriented DBMS (an *OO*DBMS), or a combination of a relational DBMS and an object-relational mapper.

An ODA based on the ideas presented here was developed as part of the Sunrise Project[1] at the Los Alamos National Laboratory (LANL). This adapter has been used by the TeleMed system [4] since mid 1995, and is currently employed by other LANL projects as well. We have implemented it for two ORBs, Orbix and VisiBroker for C++, with ObjectStore as the underlying ODBMS in both cases. Even though these implementations were aimed at a non ODMG-compliant ODBMS, we report our experience in ODMG terms whenever possible.

1.1 The Case for an ODA

ODBMSs integrate database capabilities with an object-oriented programming language. They implement *persistent memory*, a single-level store abstraction of the memory hierarchy. An ODBMS with C++ bindings provides a persistent address space for C++ objects, with heap-style allocation/deallocation. ODBMS programmers manipulate persistent C++ objects in the same way they manipulate objects in the transient heap.

Nevertheless, a CORBA server implemented in C++ cannot simply place in persistent memory the objects it implements. To have the status of a CORBA object, a C++ object must be registered with the ORB, which keeps a per-server-process table of active objects. The details of how C++ objects are registered as CORBA objects are not fully specified by the current release of CORBA.[2] In existing ORBs, CORBA objects are registered upon creation. The following approaches are currently used by ORB implementations:

1. A server may create CORBA objects only via calls to the ORB, usually to the function `BOA::create`.

2. A server can instantiate CORBA objects directly. The constructor of a CORBA object executes IDL-generated code that registers the object with the ORB.

On the other hand, the ODBMS provides its own overloaded form of operator `new`. It requires persistent objects to be created by this operator. If the ORB enforces approach 1 above, then there is clearly no way of placing a CORBA object in persistent memory. If the ORB supports approach 2, one could naively use the overloaded form of operator `new` to instantiate "persistent CORBA objects". This would not work, because the constructor of a persistent object is invoked only when the object is added to the database: "persistent CORBA objects" stored by other processes (including previous runs of the same server program) would not be registered with the ORB. As far as the ORB is concerned, these objects would not be active — no requests would be delivered to them.[3]

To make the ORB and the ODBMS work together, an additional component is necessary. Driven by incoming requests, such a component should activate objects that lie dormant in persistent memory. To allow on-demand activation of dormant objects, it must ensure that object references handed out to CORBA clients contain information on the location of the corresponding objects in persistent memory. Hence this component has to be responsible for the generation and interpretation of references to persistent objects. In the OMG ORB architecture these responsibilities belong to an Object Adapter.

1.2 The Role of the ODA

The primary role of the ODA is to provide CORBA servers with an application-independent way of making CORBA objects persistent. This includes ensuring that references to persistent objects are themselves persistent. In CORBA, *persistence of object references* means that "a client that has an object reference can use it at any time without warning, even if the (object) implementation has been deactivated or the (server) system has been restarted" [7].

[1] See `http://www.acl.lanl.gov/sunrise/sunrise.html`.

[2] The underspecification of a number of server-side issues led to server portability problems [9], which the OMG is about to solve [1].

[3] CORBA distinguishes *object activation* (activation of individual objects within a server) from *implementation activation* (server activation, usually performed by ORB-provided daemons).

With persistence of object references, it makes perfect sense for a client to store an object reference for later use. References to persistent CORBA objects implemented by server X can be stored by server Y (a client of server X), thereby enabling the construction of ORB-connected multidatabases. In such a multidatabase, references to remote objects are used to express relationships between CORBA objects implemented by different servers.

Distributed transactions, in an ORB-connected multidatabase, should be supported by a TP monitor that implements the Object Transaction Service (OTS) specified by the OMG [8]. In the absence of this service, the ODA has the additional role of ensuring that operations on persistent objects are encompassed by local transactions.[4] It interacts with the ODBMS to start and commit (or abort) database transactions.

1.3 Organization of this Paper

The next section motivates and presents the general design of the ODA. Section 3 discusses implementation issues; Section 4 considers transactions; Section 5 examines the ODA interfaces and their typical usage; Section 6 mentions related work; and Section 7 presents concluding remarks.

2 Design Decisions

Our perspective is the one of a third-party implementor, with no access to ORB and ODBMS internal interfaces. Accordingly, our ODA is an add-on to the ORB's native Object Adapter (OA), rather than a replacement for it. Figure 1 shows how it fits into the integrated ORB/ODBMS environment.

Note that the ODBMS is depicted as a separate entity holding persistent objects. This representation exposes the three-tiered nature of the ORB/ODBMS environment: an object implementation — the middle tier — is at the same time a client of the ODBMS and a server to CORBA clients. For simplicity, in a subsequent figure we omit the ODBMS and represent persistent objects within the CORBA server. The reader should keep in mind that "persistence within an object implementation" is a simplified representation of the architecture in Figure 1.

[4] Several *OO*DBMSs, including ObjectStore, do not yet support the resource manager interface required by OTS. This service might also be absent simply because a particular application does not need distributed transactions.

2.1 What to Place in Persistent Memory

A CORBA object has two parts: an IDL skeleton and an user-defined part.[5] The *skeleton* consists of ORB-specific data members and member functions, all of them mechanically generated from an IDL specification. It is an instance of a *skeleton class*, a server-side dispatcher generated by the IDL translator. The user-defined part is the *implementation object*, an instance of an *implementation class* provided by the server writer. The implementation object encompasses the data members and member functions actually defined by the object implementor.[6]

The data members in the user-defined part of a CORBA object are relevant to the application, the ones in the skeleton part are relevant to the ORB only. If we employ an ODBMS to make CORBA objects persistent, we should certainly keep their implementation objects in persistent memory. Should we also place their skeleton parts in persistent memory? An obvious reason for not doing so is waste of database space, specially in the case of fine-grained objects.[7] Stronger reasons are:

ORB independence. Keeping ORB-specific data members in persistent memory ties the database to a particular ORB implementation. As ORB products evolve, these data members may change with ORB releases. Databases with ORB-specific information would then have to go through a schema evolution process.

Performance. Assuming that CORBA objects are reference counted,[8] the skeleton part of a CORBA object holds its reference count, which is updated by the primitives `duplicate` and `release`. Placing reference counts in persistent memory means encompassing these primitives by update transactions. Every operation that receives or returns a reference to a persistent object would then require an update transaction, because parameter passing involves `duplicate` and `release` calls.

Only the user-defined parts of CORBA objects should be placed in persistent memory. As the ODA activates

[5] We are not considering the case of CORBA objects implemented with the Dynamical Skeleton Interface.

[6] Terminology could be better here, as *implementation object* is easily confused with *object implementation*. The former is an instance of an implementation class, the latter is the OMG term for a CORBA server. We prefer the vocabulary adopted by [5] — *servant* for implementation object, *servant class* for implementation class — but refrain from using it, because the new Portable Object Adapter specification [1] has assigned another meaning to the word servant.

[7] Besides ORB-specific data members, the skeleton part of a CORBA object typically has a pair of hidden vbase and vtable pointers for each interface class in the object's inheritance chain up to `CORBA::Object`.

[8] Although CORBA does not specify such implementation details, most (if not all) ORB implementations keep a reference count per object.

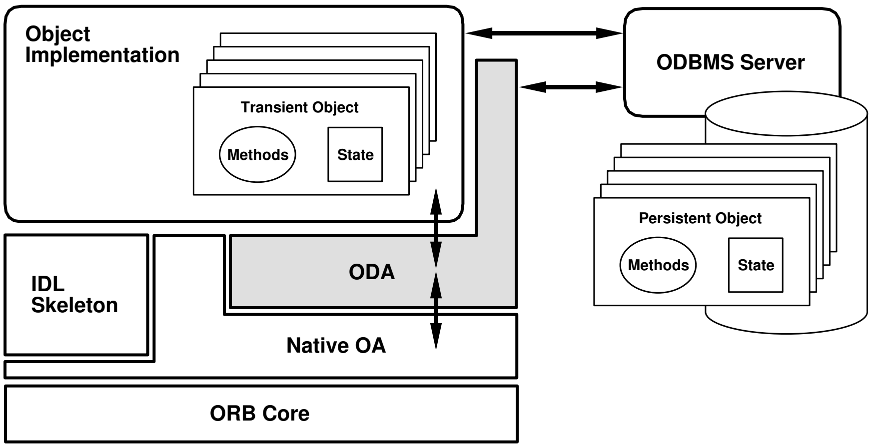

Figure 1: The Object Database Adapter.

and deactivates objects, it should dynamically instantiate and release their skeleton parts, allocated in transient memory. These observations lead us to a clear choice with respect to the relationship between skeletons and implementation objects.

2.2 Delegation, Not Inheritance

Figure 2 shows the alternatives commonly used to connect the parts of a CORBA object. In the *inheritance approach*, the object implementor derives implementation classes from IDL-generated skeleton classes. In the *delegation approach*, also known as *tie approach*, instances of IDL-generated skeleton classes are called *tie objects*, or simply *ties*. Each tie holds a reference to an implementation object to which it delegates operations. While inheritance imposes identical lifetimes to both parts of a CORBA object, delegation allows implementation objects to outlive their skeleton objects. We therefore choose delegation as the interface implementation approach supported by the ODA.

2.3 Pseudopersistence

Our decisions can be summarized as follows:

- The ODA supports persistent CORBA objects implemented with the delegation approach.
- CORBA servers keep only implementation objects in persistent memory.
- The ODA is responsible for dynamically instantiating and releasing transient ties to persistent implementation objects, so that full CORBA objects are available whenever they are needed.

Even though "persistent CORBA objects" are not fully kept in persistent memory, to their clients they appear as long-lived objects. Accordingly, we call this scheme *pseudopersistence*. In what follows, a *pseudopersistent tie*, or simply *p-tie*, is a transient tie to a persistent implementation object.

As any tie, a p-tie has a data member that specifies the implementation object to which the tie delegates operations. In a regular tie, this data member is a C++ pointer or reference. In a p-tie, it must be an ODBMS reference (`d_Ref`), for it points to an implementation object in persistent memory.

When a p-tie is instantiated, one must initialize its `d_Ref` data member. To support the instantiation of a p-tie given a CORBA object reference, the ODA embeds a `d_Ref` to an implementation object into every CORBA reference it generates. This embedding takes advantage of the `id` (also known as `ReferenceData`) field of the object reference. The `id`, an octet sequence opaque to the ORB core, contains identification information local to the server in which the CORBA reference was generated. References to p-ties are generated and interpreted by the ODA, which embeds `d_Ref`s into their `id`s.

Figure 3 illustrates the pseudopersistence scheme. A request to a dormant object arrives through the ORB core (1), causing an upcall to an ODA-provided object activation function. The `id` field of the target object reference is passed as a parameter to this function. This `id` contains a stringfied `d_Ref` to a persistent implementation object. The ODA extracts the `d_Ref` from the `id` and passes it as an argument to an instantiation function (2), which constructs the target CORBA object as a p-tie to the implementation object specified by the `d_Ref`. The incoming request then reaches the target object as an upcall through the IDL skeleton (3). At the end of the operation, another upcall to the ODA (4) causes the target object to be released.

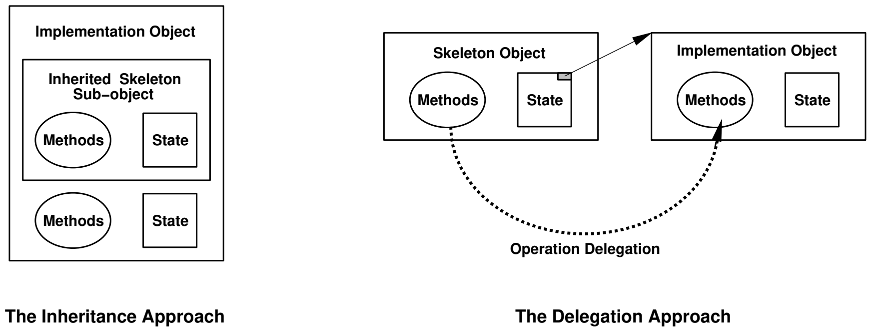

Figure 2: Interface implementation approaches.

In Figure 3, the object activation upcall happens because the target of an incoming request is a dormant object. Upcalls also happen in the case of dormant objects referenced by request parameters, or by strings passed to `string_to_object`.

Pseudopersistence should be understood in the context of the architecture in Figure 1. Persistent implementation objects do not live within a CORBA server, as the simplified representation in Figure 3 may suggest, but in a database server. Multiple CORBA servers (middle-tier processes) may be clients of a database server; they may or may not run in the same node as the database server. Moreover, a persistent implementation object may be shared by multiple p-ties, each in the address space of a different middle-tier process.[9]

3 Implementation Issues

The ODA is implemented as a library that uses and extends the services of the native OA. It requires changes on the IDL translation process, which must become ODA-aware. These changes, as well as the actions of the ODA library, are examined below.

3.1 IDL Translation Issues

- Any tie class has a data member that references the implementation object to which ties delegate operations. This data member is usually a C++ pointer or reference. In the case of a p-tie class, however, it must be a `d_Ref`.

- Code to support the management of p-ties by the ODA library must be generated within every p-tie class. In our implementation, p-tie constructors and destructors perform ODA-related actions. Moreover, each p-tie class makes available to the ODA library a static function for p-tie instantiation.

The constructor of a p-tie class embeds into the p-tie's `id` a stringfied `d_Ref` to the p-tie's implementation object. It also registers the p-tie with the ODA library; the p-tie will be eventually unregistered by its destructor. The p-tie instantiation function receives a `d_Ref` to a persistent implementation object and creates a new p-tie to this object.

Special translation requirements do not necessarily mean another IDL translator. Our ODA implementation actually employs the IDL translator provided by the ORB, complementing it with macros. The object implementor annotates the server code with ODA-defined directives, which macro-expand into p-tie class definitions. No changes are made to any files generated by the IDL translator: ODA directives are placed only in user-written files, and typically within server headers. In what follows, an *ODA-generated function* (*ODA-generated class*) is a function (class) defined by the macro expansion of an ODA directive.

3.2 ODA Actions

- The ODA library receives object activation upcalls from the native OA, forwarding each such upcall to the appropriate p-tie instantiation function.

- At the end of every operation, after any results were marshaled into a reply message, the ODA library issues `release` calls on all p-ties instantiated while the current request was being serviced.

Because the number of implementation objects in a database is potentially very large, a CORBA server can-

[9]To exemplify: consider a persistent CORBA object implemented by an Orbix server whose activation mode is *per-client-process*, and suppose that multiple clients are concurrently using this object. Every client interacts with its own middle-tier process, a distinct execution of the same server program. Each middle-tier process has a p-tie that incarnates the persistent CORBA object. All these p-ties share the same persistent implementation object, which is managed by the ODBMS.

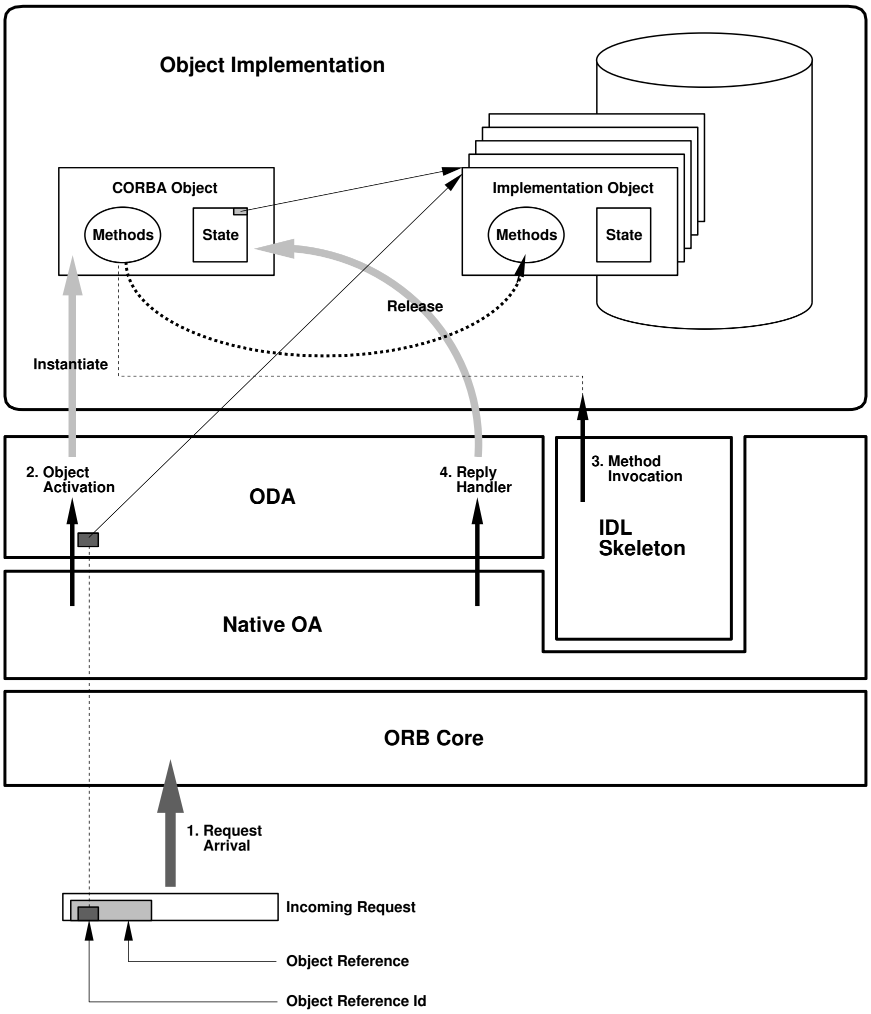

Figure 3: The pseudopersistence scheme.

not keep in-memory ties to all the persistent implementation objects it touches during its execution. The last item above addresses the need of releasing p-ties from time to time. Each p-tie is instantiated with a "net reference count" of zero — an initial reference count of one, plus a pending `release` call, to be performed by the ODA at the end of the operation. Unless the server code issues `duplicate` calls on them, p-ties have short lifetime: they exist while a request is being serviced. Whenever a discarded p-tie is needed again, an equivalent to it will be instantiated by an object activation upcall.

3.3 Caching P-ties

Releasing *all* p-ties at the end of every operation appears unreasonable, since the ODA only needs to ensure that these p-ties will be eventually released. Postponing their destruction would avoid the costs of successive p-tie re-instantiations. Our ODA implementation actually caches the last N p-ties it instantiated, where N is a configurable parameter. At the end of every operation the ODA brings the number of p-ties down to $N + \delta$, keeping the most recent ones. (The δ accounts for any `duplicate` calls that might have been issued by the server code.)

Caching p-ties makes sense if the ODBMS ensures that their `d_Ref` data members remain valid across transactions. So far we have ignored database transactions, this topic will be discussed in Section 4. Let us assume, by now, that transactions are started and committed (or aborted) by means external to the CORBA server, and that each operation is encompassed by an individual transaction.

Does a `d_Ref` from transient to persistent memory retain its validity across transactions? The ODMG standard leaves the answer to the discretion of the ODBMS implementor. In most ODBMSs, such a reference cannot be used in between transactions, but does remain valid across transactions. This being the case, the ODA should cache p-ties.

With caching, the CORBA server must have a way of forcing the removal of objects from the cache. Accordingly, the ODA provides a function that receives a `CORBA::Object_ptr` and immediately deletes the corresponding p-tie. This function, `ODA::Delete`, is intended to be called by destructors of persistent implementation objects, with the purpose of avoiding dangling p-ties.

3.4 Converting Implementation Objects to CORBA Objects

The ODA must provide the server code with the means for obtaining a CORBA object given its implementation object. For each association

(interface, implementation_class)

there is an ODA-generated function that takes a `d_Ref` to an implementation object and returns a reference (of type *interface*`_ptr`) to the corresponding CORBA object. To avoid multiple p-ties to an implementation object, this function is not implemented as a mere call to p-tie instantiate. It first checks it if a p-tie to the implementation object already exists in the server's address space, then it returns a `duplicate`d reference to either an existing p-tie or a newly instantiated one.

A non-standard `bind` function, present in various ORBs, could be used to perform the check mentioned above. Given a `d_Ref` to an implementation object, one would convert it to string and obtain an `id`, which would then be passed as an argument to `bind`. The ODA does not use this approach. Instead, it keeps pairs

(`d_Ref`*, p-tie_address)*

in its own table of active p-ties, which it hashes by `d_Ref`s with a hash function provided by the ODBMS.

3.5 Non-standard ORB and ODBMS Features Employed

The ODA relies on the delegation approach, which is mentioned — but not mandated — by the current revision (2.0) of CORBA. Orbix and VisiBroker are examples of commercially available ORBs that support delegation. Both admit direct instantiation of ties, automatically registering newly instantiated ties as active CORBA objects. The new Portable Object Adapter (POA) specification [1] fully standardizes the delegation approach. It requires compliant ORB implementations to support both the inheritance and the delegation approach.

Because CORBA 2.0 describes object activation in very general terms, existing ORB implementations vary widely on their support to object activation. The ODA builds upon the native OA's object activation capabilities. Its Orbix implementation uses a `LoaderClass` instance; the VisiBroker implementation uses an `Activator`. The new POA specification completely defines object activation. With the POA, future ODA implementations can employ a standard facility, the `InstanceActivator` interface.

Various ORBs provide non-standard "event handling" or "request/reply intercepting" mechanisms. The ODA needs such a facility both to release p-ties and to manage database transactions in the absence of OTS (see Section 4). Its Orbix implementation uses a `Filter`; the VisiBroker implementation uses an `EventHandler`. The OMG has recently introduced *request level interceptors* [10] as an extension to the ORB core, and is actively working to complete the specification of this facility.

From the ODBMS, the ODA requires a means of converting `d_Ref`s to strings and vice-versa. Although supported by many ODBMSs, this feature is not in the ODMG standard. Caching of p-ties requires more: `d_Ref`s must remain valid across transactions.

4 Transactions

Any access to persistent memory has to be performed within a transaction. Leaving to implementation objects the responsibility of starting and committing (or aborting) transactions is not an option, because accesses to persistent memory happen both before and after these objects' methods are called:

- In order to delegate an operation to its implementation object, a p-tie must access persistent memory. The p-tie must dereference its `d_Ref` data member, which points to persistent memory.
- Marshaling of operation results into a reply message may involve accesses to persistent memory.

Usage of OTS [8] would ensure that not just the user-provided implementation code, but also request dispatching and parameter marshaling code, would be executed within transactions. Since OTS interacts directly with the local resource manager (the ODBMS), transactions would be started and committed (or aborted) by means external to the CORBA server.

If OTS is absent, the ODA must take the responsibility of starting and committing (or aborting) *local* transactions. Not with the aim of performing distributed two-phase commit, but just to ensure that a transaction will be active whenever an operation is dispatched, and will remain active till the operation results are marshaled into a reply message. We did not have OTS, so this was our scenario.

4.1 Support to Local Transactions

The ODA manages local transactions by employing ORB-specific “event handling” or “request/reply intercepting” facilities. Its default transaction mode is *transaction per operation*: an “incoming request pre-marshal” handler starts a transaction as soon as a request arrives, an “outgoing reply post-marshal” handler ends the transaction just before the reply is sent. An operation implementation may specify if the current transaction will be committed or aborted at the end of the operation. By default, the ODA commits the transaction. Under control of the server code, the ODA may also switch to another transaction mode, which allows multiple operations to be grouped into a single transaction.

Because ObjectStore requires the transaction type (read-only or update) to be specified when a transaction starts, update operations must be registered with the ODA. Registration of update operations is typically done by the server mainline. By default, the ODA starts read-only transactions. In the case of operations previously registered as update operations, it starts update transactions.

5 ODA Interfaces and Usage

The CORBA server interacts with the ODA through a very small API. Besides ODA-generated functions that return an *interface*`_ptr` given a `d_Ref` and vice-versa, there are just a few static functions available to the server code:

- `ODA::initialize`
- `ODA::register_update_ops`
- `ODA::Delete`
- `ODA::multi_op_transaction_mode`
- `ODA::abort_transaction`
- `ODA::commit_transaction`

Note that there is no specific function to create or activate a persistent CORBA object: object activation may occur as a side effect of the conversion of a `d_Ref` into CORBA object reference.

Given an interface class `X` and an implementation class `X_i` to which `X` delegates operations, the function

```
X_ptr ODA_X_i_to_X(const d_Ref<X_i>&);
```

translates a `d_Ref<X_i>` into the corresponding `X_ptr`. This function, defined at the file scope, is generated by the ODA directive that “ties together” `X` and `X_i`. A member function of the ODA-generated p-tie class performs the reverse translation (to `d_Ref<X_i>`).

The ODA is not an intrusive presence in the programming environment. In our experience, the vast majority of ODA calls is performed to obtain an *interface*`_ptr` from a `d_Ref`. Except for these, ODA calls are relatively rare in the server code. `ODA::initialize` is called once, by the server's mainline. Calls to `ODA::register_update_ops` typically appear in the server's mainline only, and would not be necessary in the case of an ODMG-compliant ODBMS. `ODA::Delete` is invoked from destructors of persistent implementation objects. In the default transaction mode, user-provided methods do not normally call transaction management functions.

5.1 Server Organization

Persistent relationships between CORBA objects within a server are actually realized by relationships between their corresponding implementation objects. When traversing database relationships or performing a database query, the server code deals only with persistent implementation objects, not with full CORBA objects. Such a traversal or query is therefore executed at ODBMS speeds. Consider, for example, the case of an operation that performs a search for a particular object within a collection of objects. The whole search is performed at the ODBMS level, without CORBA-activating any of the objects of the collection. Its result, a `d_Ref` to particular implementation object, is then converted to CORBA object reference and passed back to the client. When the server code calls the ODA to perform such a conversion, it obtains a `duplicated` reference to a CORBA object managed by the ODA. Whether this object was just activated or was already in the ODA cache is irrelevant to the server code, which in either case assumes the responsibility of releasing the reference.

Persistent relationships between CORBA objects in different servers are realized via stringfied CORBA references stored in persistent memory. These references must be explicitly converted back to its native form for usage. Note that any database containing CORBA object references is ORB-dependent, because these references are ORB-dependent. ORB independence is lost when we move on to an ORB-connected multidatabase.

5.2 Inheritance Issues

Consider the following IDL interfaces:

```
interface X { ... };
```

```
interface X1 : X {
    ...
};

interface X2 : X {
    ...
};

interface Y {
    readonly attribute X x;
    ...
};
```

Interface X defines operations that are common to both X1 and X2. Attribute x of Y has interface type X, but its most derived interface is either X1 or X2.

A natural organization for the corresponding persistence-capable implementation classes[10] would be:

```
class X_i : public d_Object {
    // abstract class
    ...
};

class X1_i : public X_i {
    ...
};

class X2_i : public X_i {
    ...
};

class Y_i : public d_Object {
public:
    X_ptr x();
    ...
private:
    d_Ref<X_i> x_i;
    ...
};
```

`X_i` is an abstract class: any instance of this class is an instance of either `X1_i` or `X2_i`. Class `Y_i` holds an ODBMS reference to an instance of `X_i` in its private data member `x_i`. The attribute accessor `Y_i::x()` returns a CORBA reference to the object whose implementation is `x_i`.

Note, however, that there is no ODA-generated function that takes a `d_Ref<X_i>` and returns an `X_ptr`. The ODA provides this conversion function only when the interface skeleton and the implementation class are tied together by delegation. This is never the case for an inherited implementation class, such as `X_i`. In the example above, there are ODA-generated conversion functions from `d_Ref<X1_i>` to `X1_ptr` and from `d_Ref<X2_i>` to `X2_ptr`.

ODA users solve this problem by defining a virtual member function, say `get_X_ptr()`, in class `X_i`. This function, declared as pure virtual in `X_i`, is redefined by the derived classes `X1_i` and `X2_i` as below:

```
X_ptr X1_i::get_X_ptr() {
    return ODA_X1_i_to_X1(d_Ref<X1_i>(this));
}

X_ptr X2_i::get_X_ptr() {
    return ODA_X2_i_to_X2(d_Ref<X2_i>(this));
}
```

If the inheritance chain were longer, all abstract implementation classes would define `get_X_ptr()` as pure virtual.

6 Related Work

Work recently concluded at the OMG, in the context of the ORB Portability Enhancement RFP [9], has resulted in a Portable Object Adapter [1] that will reduce the ODA dependencies on non-standard ORB features. Earlier ORB portability proposals [5, 3] included a Server Framework Adapter (SFA) and an ODMG model for SFA. Our pseudopersistence scheme is essentially a realization of the ODMG model for SFA, as outlined in the Appendix C of [3].

A number of ORB and ODBMS vendors has announced plans for the integration of their products; some of these integrated solutions are already being delivered. Probably the first one was Iona Technologies's Orbix+ObjectStore Adapter (OOSA) [6], whose beta release became available by late 1995. Since then, Iona has integrated Orbix with Versant, and has announced plans for integrating Orbix with O2 and with Persistence.

Iona's OOSA takes advantage of the particular way CORBA objects are laid out by the ORB. In Orbix, not all data encapsulated by a `CORBA::Object` instance appears directly in its data members. Instead, a data member of `CORBA::Object` points to an auxiliary object. Some of the "logical" data members of `CORBA::Object` are actually in this auxiliary object. The reference count is one of them.

Unlike the ODA, which stores only implementation objects, OOSA actually stores CORBA objects in ObjectStore databases. A CORBA object, however, is not

[10] We adopt the convention of naming implementation classes by appending an "`_i`" to the corresponding interface names.

stored in their entirety: to avoid the performance penalty of having reference counts in persistent memory, OOSA does not store the auxiliary object in the database. Instead, it dynamically instantiates auxiliary objects as persistent CORBA objects are made available in ObjectStore's client cache. When such an auxiliary object is instantiated, the corresponding CORBA object is inserted into the per-process table of active objects maintained by Orbix. This approach allows persistent CORBA objects to be implemented either by inheritance or by delegation. It also allows object relationships to be expressed in terms of CORBA objects, not just in terms of implementation objects. Its disadvantages are some waste of database space, ORB-dependent databases, and the performance penalty of object activations triggered by database accesses.

7 Concluding Remarks

We have presented the design and implementation of an ODA that allows execution of database traversals and queries at the full speed of the underlying ODBMS. Only what needs to be persistent is kept in persistent memory; ODA users are not forced to store ORB-specific information persistently. Databases are ORB-independent unless the user explicitly places ORB-specific data (such as stringfied object references) in persistent memory. Finally, the ODA design appears to be general enough to be applicable to any ODBMS. ObjectStore's virtual memory-based architecture makes it different from all other ODBMSs in many aspects. That the ODA design can be described in ODMG terms, and yet be implemented for ObjectStore, is strong evidence of its applicability to any ODBMS.

The ODA's pseudopersistence scheme appears to be an optimal solution for integrated ORB/ODBMS environments in which object relationships are mostly confined within a CORBA server. In such a scenario, there is no reason to express database relationships at the CORBA level, as they are much more efficiently realized at the level of implementation objects.

The motivation for representing database relationships at the CORBA level might arise in the context of an ORB-connected multidatabase with many cross-server references. Expressing persistent relationships between objects in different servers via stringfied CORBA references placed in persistent memory may be inconvenient in this case. Consider, for example, a situation in which it would be desirable for a server to have a persistent and homogeneous collection of object references, whose elements may refer to either local or remote objects. This is not possible in the pseudopersistence scheme. Instead of a uniform collection, two distinct sub-collections must be used: one with `d_Refs` to local implementation objects, other with stringfied CORBA references to remote objects. Intra-server references and inter-server references could be unified if the Object Adapter provided support for persistently representing *both* at the CORBA level. To be useful, this unification should allow transparent use of stored CORBA references to invoke methods on possibly remote objects. Note, however, that incurring the cost of such a unification — the performance penalty of expressing intra-server references at the CORBA level — makes sense only if cross-server references occur much more than intra-server references.

References

[1] BEA, DEC, Expersoft, HP, IBM, ICL, IONA, Novell, SunSoft, and Telefónica I+D. *ORB Portability Joint Submission, Draft 14*. OMG Document orbos/97-04-04, April 1997.

[2] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, 1996.

[3] DEC, Expersoft, HP, IBM, ICL, IONA, Novell, SunSoft, and Telefónica I+D. *ORB Portability Joint Submission, Draft 5*. OMG Document orbos/96-12-02, December 1996.

[4] D. W. Forslund, R. L. Phillips, D. G. Kilman, and J. L. Cook. Experiences with a distributed virtual patient record system. *Journal of the American Medical Informatics Association, Symposium Supplement*, 1996.

[5] HP, IBM, Novell, and SunSoft. *Server Framework Specification (ORB Portability Submission)*. OMG Document orbos/96-05-03, May 1996.

[6] Iona Technologies. *Object+ObjectStore Adapter — Beta Release Documentation*. Dublin, Ireland, 1995.

[7] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. Revision 2.0, July 1995.

[8] Object Management Group. *CORBAservices: Common Object Services Specification*. Revised Edition, March 1995. Updated November 1996.

[9] Object Management Group. ORB Portability Enhancement RFP. OMG Document 95-06-26, June 1995.

[10] Object Management Group. *CORBASecurity*. Version 1.1, OMG Document Numbers 96-08-03 through 96-08-06, July 1996.

[11] Object Management Group. Persistent Object Service, version 2.0 — Request For Proposal (Draft). OMG Document orbos/97-04-07, December 1997.

[12] F. C. R. Reverbel. *Object Database Adapter Programmer's Guide and Reference Manual*. Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos, NM, August 1996.

[13] F. C. R. Reverbel. *Persistence in Distributed Object Systems: ORB/ODBMS Integration*. PhD thesis, University of New Mexico, Computer Science Department, Albuquerque, NM, May 1996.

[14] S. Vinosky. CORBA: Integrating diverse applications within heterogeneous environments. *IEEE Communications*, 14(2), February 1997.