# Persistence in Distributed Object Systems: ORB/ODBMS Integration[1]

Francisco Reverbel[2]

Ph.D. Dissertation Presented to the Computer Science Department of the University of New Mexico

April, 1996

[1]This research was performed at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545, as part of the Sunrise Project.

[2]During the development of this work the author was on leave of absence from the Computer Science Department of the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP), São Paulo, Brazil, and was partly supported by a fellowship from the National Research Council of Brazil (CNPq).

*committee signatures page*
*(provided by the university)*

*title page*
*(provided by the university)*

*To Martha and our children*

# Acknowledgments

This dissertation would have been impossible without the encouragement and support of many people at different places. I cannot possibly thank everyone individually, but would like to express my gratitude to them all.

**At the University of New Mexico**

I have been very fortunate to have had Professor Barney Maccabe as my Ph.D. advisor. Barney introduced me to the field of massively parallel computing, guided my early work in this field, and continued counseling me even after I decided to redirect my research to the area of distributed systems. I am grateful to Barney not only for his technical advice, but also for the encouragement and support he gave me when they were most needed, not to mention his patience with a student who took too long to settle on a dissertation subject.

I would like to thank Professors John Brayer, Charles Crowley, and Gregory Heileman for serving in my committee and for their helpful comments and suggestions. And would also like to extend my thanks to other faculty members of the Computer Science Department. Professors Bernard Moret, Stephanie Forrest, Henry Shapiro, and Paul Helman indirectly contributed to this work by teaching some of the excellent classes I have attended. Professor Edward Angel helped me as Graduate Advisor, during my initial semesters at the department, and by recommending me to the Sunrise Project.

Joann Buehler, Graduate Secretary of the Computer Science Department, deserves special thanks for her thoughtfulness and efficiency. My last weeks as a Ph.D. student, which were quite hectic, would have been yet harder without her help.

Many thanks to Martin Mueller, Ksheerabdhi Krishna, and Tom Claus, fellow graduate students, for their friendship, encouragement, and support. Martin had a decisive influence on my graduate career: he pointed to me a Los Alamos project just

initiated, in which I found the dissertation topic I was looking for.

**At the Los Alamos National Laboratory**

Dave Forslund, Deputy Director of the Advanced Computing Laboratory and Principal Investigator of the Sunrise Project, inspired much of this work by sharing with me his vision of distributed object computing. I am grateful to Dave for his stimulating insights and for the excellent research environment at the laboratory.

Dick Phillips, who led the TeleMed project (one of the Sunrise sub-projects), suggested to me ORB/ODBMS integration as a Ph.D. topic. I thank him for the good hint and for believing that I could provide the integrated ORB/ODBMS environment TeleMed needed.

I have learned a lot from the members of the Sunrise/TeleMed team: Bob Tomlinson, Ron Daniel, Mohamad Ijadi, and Juhnyoung Lee, among others. I am also grateful to Juhnyoung for his encouragement and support. Juhnyoung and I have collaborated closely on applying ORB/ODBMS integration techniques to relational systems; a solid friendship developed while we worked together.

Besides answering all my questions on how to draw pictures and make plots, Pat McCormick let me use his SGI workstation to draw the pictures that appear in this dissertation. Until I was done with the drawings, Pat never forgot to log out from his computer at the end of each day (rather than simply locking the screen), so that I could use it after hours. Just to draw pictures, of course — his machine is great, but I still prefer the keyboard of my SPARCstation.

Finally, many thanks to Dave Kilman. His guitar helped me to remain sane throughout months of incredibly hard work.

**At home**

My mother-in-law, Gioconda Monteiro, came three times to the USA while my wife and I were both laboring over our graduate duties. She spent many months in this country, just to help us with the children — I cannot thank her enough.

And thank you Martha, for everything. Without your love I would not have finished this dissertation.

**Institutional Supporters**

This research was performed at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545. It was carried out as part of the Sunrise Project, an effort of the Los Alamos National Laboratory with the goal of providing a nationally scalable environment which facilitates the dynamic assembly of distributed industrial and scientific applications.

For the duration of this work I was on leave of absence from the Computer Science Department of the Institute of Mathematics and Statistics of the University of São Paulo (IME-USP), São Paulo, Brazil, and was partly supported by a fellowship from the National Research Council of Brazil (CNPq).

*abstract title page*
*(provided by the university)*

# Abstract

This dissertation discusses the realization of object persistence, in a CORBA-based distributed system, through the integration of the Object Request Broker (ORB) with an Object Database Management Systems (ODBMS). Three approaches to persistence of CORBA objects are described. The first one, *pseudopersistence*, makes object references persistent, but not transparently storable. The others, *smart pointer-based persistence* and *virtual persistence*, provide transparent storability to object references, both in the case of references to local objects and in the case of references to remote objects. We stress the ORB and ODBMS features each approach depends upon, and point out the ones not fully standardized at the present time.

A software component, the Object Database Adapter (ODA), is responsible by the integration of an ORB with an ODBMS. We include a report on the use of an ODA in a real-world telemedicine application, a distributed, object-oriented, CORBA-based patient record system developed at the Los Alamos National Laboratory.

A performance evaluation of the aforementioned approaches to persistence of CORBA objects is presented. Our quantitative results reveal the good performance of pseudopersistence, the effectiveness of active object caching, the promising performance of smart pointer-based persistence, and the penalty paid for using virtual persistence with the single commercially available ODBMS to which this approach is applicable.

# Contents

# List of Figures

# List of Tables

# Introduction

As Birrel *et al.* pointed out in [3], the object-oriented programming methodology applies beautifully to distributed computing. Since object-oriented programming enforces encapsulation, clients can only access the state of an object via the object's methods. The method calls are therefore a natural place to insert the communication needed in a distributed environment. Even some words of the object-oriented terminology ("message", "receiver") reveal the potential affinity between object-oriented programming and distributed computing. At the root of this affinity is the concept of separation, fundamental to both disciplines: in one of them, it appears as the logical separation among encapsulated objects; in the other, as the physical separation among distributed entities.

Although research systems based on this observation started to appear by the mid 1980s, distributed objects have not yet attained mainstream status. More than the many technical issues involved in the design of a distributed object environment[1], the lack of a standard for such environments prevented their widespread use. This situation is now changing with the emergence of two competing standards: CORBA, the Object Request Broker (ORB) architecture specified by the Object Management Group (OMG), and COM, the Component Object Model upon which Microsoft based its Object Linking and Embedding (OLE) infrastructure.

Nearly all major software and hardware vendors are backing the OMG architecture, Microsoft being the single and notable exception. Due to Microsoft's position in the software industry, however, it is likely that COM will become an alternative standard — a *de facto* standard for personal computer platforms[2]. CORBA prevalence is

[1]See the overview section of [3] for a brief discussion of some of these issues.

[2]COM is Microsoft's proprietary solution, not a formal standard. Only its non distributed (single-

expected on all other platforms, and especially on POSIX systems. It appears that both standards are here to stay, and will be coexisting in the foreseeable future. The present work addresses CORBA [37], because:

1. several ORB implementations are available now;
2. as the *de jure* standard, CORBA is an open and cross-platform solution;
3. most neutral object technology experts agree that the OMG architecture is technically superior to OLE/COM.

# Our Subject

In the OMG model for distributed object systems [40], the ORB is the software component through which objects transparently make requests and receive responses. It provides interoperability between applications written in different languages and running on different machine architectures, seamlessly interconnecting multiple object systems in heterogeneous distributed environments.

In many situations, objects should outlive the processes that implement them. These situations include most practical scenarios in which distributed objects are desirable: in real-world applications, persistence is almost always a requirement. While CORBA does not support object persistence directly, its open architecture includes an *object adapter*, an intermediary between an object implementation and the ORB core. Object persistence can then provided by the object adapter.

The CORBA specification defines a Basic Object Adapter (BOA) that does not provide object persistence. As an example of another object adapter that might be useful, CORBA briefly mentions an 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 [7], 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.

---

machine) version is currently available. Network COM will not be shipping before late 1996, with Cairo's Network OLE.

An Object Database Adapter allows object implementations to be written in the database programming language of the ODBMS, a language that incorporates persistence into the programming environment. The object implementation — a CORBA server — is still responsible for managing the persistent state of the objects it implements, but the object implementor's task is much simpler in the programming environment provided by ODBMS. Besides persistence, other database features — data consistency in the presence of concurrent accesses, crash recovery, and so forth — are available to the object implementation.

In existing database systems, regardless of their data model (object-oriented or relational), database clients must have some knowledge of the database schema. In the case of an object-oriented DBMS, clients need to know the object layout. In a relational DBMS, views can be used for data independence. But relational clients still need to know the external (view level) schema. By contrast, ORB/ODBMS integration makes database objects accessible to CORBA clients without exposing the database schema to these clients. The data members and the layout of a persistent CORBA object remains private, only its interface (a set of methods) is made public. This is specially interesting for web browser access to databases. With the integration of the Java language ([11], [55]) into the ORB environment, Java applets can interact with persistent CORBA objects through domain-specific interfaces, without any knowledge of how the objects are actually stored.

ORB/ODBMS integration leads to ORB-connected multidatabases. Together with OTS, the Object Transaction Service[3] specified by the OMG in [39], it enables the construction of distributed object databases that are truly heterogeneous, even with respect to the DBMS software running on the various database server nodes.

## Storing CORBA Objects in an Object Database: The Issues

The gap between object access times in the ORB and in the ODBMS environment is the first issue an Object Database Adapter must address. Because CORBA clients access objects via remote method invocation, access times for CORBA objects are expressed in milliseconds. Because ODBMSs keep an object cache at the client side,

[3]OTS implements the two-phase commit protocol [12] in a CORBA environment. Its model can be viewed as an object-oriented extension of the X/Open DTP model [60].

ODBMS clients can access objects much faster: access times for ODBMS objects are typically expressed in microseconds.

Due to the this performance gap, an Object Database Adapter cannot force all accesses to the objects in a database to be made through the ORB remote method invocation mechanism. In the case of a large collection of very small objects, the overhead would be unacceptable. Instead, the ODA should let the object implementor choose a suitable subset of database objects, presumably the higher level ones, to be accessed as CORBA objects. Since this subset may still be large, individual registration of its objects with the ORB is not practical. The ODA should allow a subset of database objects to be accessed through the ORB, without requiring an explicit registration call for each object.

Moreover, in the common case of an ODBMS that adds database features to C++, an object implementation cannot simply store in an object database the CORBA objects it implements:

- It would be a waste of database space: a C++ CORBA object has ORB-specific data members that should not be stored. It typically also has a pair of hidden `vbase` and `vtable` pointers for each interface class in its inheritance chain up to `CORBA::Object`.

- More importantly, the CORBA operations `duplicate` and `release` update the object's reference count. If the reference count were actually stored in the database, every operation on the object would have to be performed within an update transaction, because `duplicate` and `release` appear everywhere.

- Yet more importantly, ORB implementations keep a per-process table of active objects: a new entry is inserted into this table whenever the constructor of a CORBA object is invoked by the corresponding process. In a C++-based ODBMS, however, the constructor of a persistent object is only invoked when the object is added to the database. As far as the ORB is concerned, CORBA objects stored by other processes (including previous runs of the same program) would not be active.

An Object Database Adapter has to solve these problems, ideally in a way that makes persistent CORBA objects appear exactly like ordinary database objects. As much

as possible, object implementors should be unaware that a persistent CORBA object does not really live in the database.

## Persistence of CORBA Object References

Besides providing persistence to CORBA objects, the Object Database Adapter must also provide persistence to the corresponding object references. 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" [37].

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) and vice-versa, 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. If distributed transactions are needed, they can be supported by a TP monitor that implements the Object Transaction Service.

## Storability of CORBA Object References

Because an object reference is opaque and ORB-dependent, CORBA provides operations that convert an object reference to string and vice-versa. Object references are stored as strings; upon retrieval they must be converted back to their native form.

Translation to and from string format provides maximum flexibility, allowing object references to be kept in any media. In an ODBMS environment, however, object storage and retrieval are transparent to the programmer. The need for explicit translation of stored references does not prevent the construction of ORB-connected multidatabases, but is unnatural in the context of an object database: ODBMS users expect stored references that behave like any other database object.

*Transparent storability* of CORBA object references is yet another desirable feature of an Object Database Adapter. It eliminates the need for explicit object reference conversions, both before storage and after retrieval, and allows transparent use of stored references to invoke methods on possibly remote objects.

## Scope of and Organization of this Dissertation

According to Cattell [6], "more work in both standards and implementation is needed to realize the integration of ODBMSs and OMG ORBs". In this dissertation we present:

- A discussion of the design issues involved in integrating ORBs and ODBMSs.

  Our experience designing and implementing Object Database Adapters is conveyed here. We have implemented two approaches to persistence of CORBA objects. The first one, which we call *pseudopersistence*, makes object references persistent, but not transparently storable. The second approach, which we call *virtual persistence*, provides transparent storability to object references, both in the case of references to local objects and in the case of references to remote objects. These approaches, as well as a smart pointer-based realization of transparently stored references, are discussed in considerable detail. We stress the ORB and ODBMS features each approach depends upon, and point out the ones not fully standardized at the present time.

- A report on the utilization of an Object Database Adapter in a real-world distributed application.

  We discuss ODA usage in TeleMed, a distributed, object-oriented, CORBA-based telemedicine system developed at the Los Alamos National Laboratory.

- A performance evaluation of ORB/ODBMS integration approaches.

  Our quantitative results reveal the good performance of pseudopersistence, effectiveness of object caching, the promising performance of smart pointer-based realizations of transparently stored references, and the penalty paid for using virtual persistence with the single commercially available ODBMS to which this approach is applicable.

The remainder of this dissertation is organized as follows: Chapters 1 and 2 review background material on ORBs and ODBMSs. Chapter 3 discusses the reasons for integrating an ORB with an ODBMS, the role of an Object Database Adapter, and ODA design issues. Chapter 4 is a report on our ODA implementation and its utilization in the TeleMed project. Chapter 5 presents a performance evaluation of different

approaches to the integration of an ORB with an ODBMS. Finally, Chapter 6 contains our concluding observations and summarizes the contributions of this work.

# Chapter 1

# The ORB architecture

A full description of the ORB architecture is beyond the scope of this dissertation. This chapter presents an overview of its characteristics and discusses in more detail the aspects relevant to ORB/ODBMS integration. For additional information, the reader is referred to the CORBA specification [37].

The Object Management Group (OMG) is a consortium dedicated to promoting and standardizing object technology for the development of distributed computing systems. It was formed in 1989 by eight companies, including Hewlett-Packard and Sun Microsystems. OMG now has over 600 members, comprising nearly all major software and hardware vendors, many large end-users, and several research institutions.

In the OMG model for distributed object systems ([40], [51]), an *object* is an identifiable, encapsulated entity that supports a set of operations. The execution of such an operation is a *service* that the object provides upon a client's request. A *client* of a service is any entity capable of requesting the service. Objects are typical clients of services: in processing a service request, an object may use services provided by other objects. Programs that do not realize objects in the sense of the OMG model — end-user applications that are not service providers, for example — can also be clients of services. Clients of services interact with the objects that provide the services through well-defined interfaces. The *interface* of an object is the set of operations a client may request of the object. All objects satisfying a particular interface have the same type, the *interface type* associated to that interface.

Concrete realizations of this model are based upon the Object Request Broker

(ORB) architecture, as defined by the OMG in the Common Object Request Broker Architecture (CORBA) specification [37]. The main components of CORBA are:

- the Interface Definition Language (IDL), in which object interfaces are described;
- the ORB core, the "communication backbone" of a distributed object system;
- an intermediary between the ORB core and object implementations, the *object adapter*, which is the primary means for an object implementation to interact with the ORB. A particular object adapter, the Basic Object Adapter (BOA), is available in every CORBA implementation. The BOA interacts with object implementations through an interface designed to support a wide variety of object implementation requirements.

These are not the only elements of the ORB architecture. Among its other components, CORBA includes an *interface repository*, a storage place for information on object interfaces, and an *implementation repository*, a storage place for information on object implementations.

## 1.1 The Interface Definition Language

The interfaces provided by objects and used by clients are defined in OMG IDL, a purely descriptive language, whose purpose is *language interoperability*. All objects that may receive requests through the ORB must have their interfaces defined in OMG IDL. The services provided by any such object can be requested by a client written in a language different from the one in which the object was implemented. Clients and object implementations are not written in OMG IDL, but in languages for which mappings from OMG IDL have been defined. Standard mappings from OMG IDL to C, C++, and Smalltalk were specified by the OMG in [37]. The standardization of a mapping to Ada 95 was also completed. Work is in progress on mappings to COBOL, Java, Objective-C, Eiffel, and Common Lisp with CLOS.

An interface definition written in OMG IDL completely specifies the interface operations. It provides information on input parameters, output parameters, and return result of each such operation, as well as information on which exceptions

may be raised when the operation is invoked. Figure 1.1 presents an OMG IDL specification of the interfaces to a bare-bones repository of medical records. This example shows how constants, types, exceptions, and interfaces are defined.

The strong influence of C++ on the OMG IDL syntax is apparent from Figure 1.1. An IDL interface looks like a C++ class; its operations are similar to member functions, and its attributes resemble data members. Unlike the data members of C++ classes, however, IDL interface attributes have no memory locations set aside for them, nor do they specify how object implementations actually store data. An attribute is logically equivalent to a pair of accessor methods; one to retrieve and one to set the value of the attribute. An attribute declared as `readonly` corresponds to a single accessor method — the one that retrieves the value of the attribute.

Data types in OMG IDL also resemble C++ types: besides interface types, there are basic types (`char`, `boolean`, `octet`, integer and floating point types), constructed types (`struct`s, `union`s, and `enum`s), template types (`sequence`s and `string`s), and array types.

IDL operations can take `in`, `out` and `inout` parameters; only `in` parameters are shown in Figure 1.1. Operation parameters and return results can have any IDL data type, including interface types. Referring to Figure 1.1, the operation `AddReport` of the `Patient` interface takes as its input parameter an object whose interface is `Report`. The operation `GetPatient` of the `Repository` interface returns an object whose interface is `Patient`.

*Interface inheritance* is supported by OMG IDL; its syntax is similar to the one used in C++ for class inheritance. The `Patient` and `Report` interfaces in Figure 1.1 are both derived from the `RepositoryData` interface. A derived interface inherits attributes and operations from its base interfaces; multiple inheritance of interfaces is allowed. Inherited attributes and operations may be redefined in the derived interface. Name resolution is done with the "`::`" operator, as in C++.

To prevent name clashes, OMG IDL has a `module` construct that defines a scope for identifiers. Examples of its usage appear in the CORBA specification itself — all names introduced by CORBA are defined within a `CORBA` module. IDL descriptions of CORBA-defined interfaces will be presented in Sections 1.3 and 1.4.

Interface specifications written in OMG IDL are processed by IDL compilers. An IDL compiler translates IDL source specifications to a target programming language

```
// constant definition
const unsigned short MAX_NAMELEN = 64;

// type definitions
typedef string<MAX_NAMELEN> Name;
typedef string Date;

// exception definition
exception reject {
    string reason;
};

// forward interface declarations
interface Patient;
interface Report;

// interface definitions
interface Repository {
    readonly attribute long number_of_patients;
    Patient AddPatient(in Name name, in Date dob) raises(reject);
    Patient GetPatient(in Name name) raises(reject);
    sequence<Name> GetPatientList() raises(reject);
    Report CreateReport(in string text) raises(reject);
};
interface RepositoryData {
    readonly attribute Name creator;
    readonly attribute Date date_created;
};
interface Patient :  RepositoryData {
    readonly attribute Name patient_name;
    readonly attribute Date patient_dob;
    readonly attribute long patient_id;
    void AddReport(in Report report) raises(reject);
    Report GetReport(in long report_id) raises(reject);
    sequence<Report> GetReportList() raises(reject);
};
interface Report :  RepositoryData {
    readonly attribute long id;
    attribute string text;
};
```

Figure 1.1: IDL specification of a bare-bones repository of medical records.

for which a mapping from OMG IDL is defined. IDL compilers for C, C++, Smalltalk, Java, and Objective-C are currently available in existing ORB systems. The results of an IDL compilation are:

- target programming language code for *client stubs*;
- *implementation skeleton* code, also in the target programming language. This is the ORB architecture's counterpart of the server stubs of RPC systems.

The IDL compiler may optionally translate interface specifications into a format suitable for use at runtime. Specifications translated into this format are stored in the interface repository, which can be dynamically queried by programs.

## 1.2 The Structure of an ORB

In a CORBA-based distributed object system, each networked machine runs ORB software. Figure 1.2 shows the overall structure of the ORB software on an individual machine[1]. The interfaces to the ORB are depicted as striped boxes, and the arrows

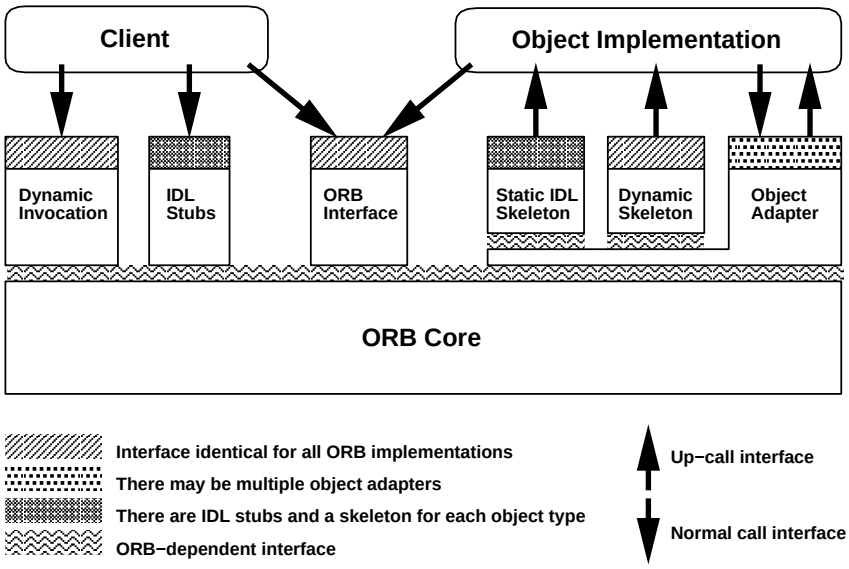



Figure 1.2: ORB structure and interfaces.

[1]Some parts of the ORB may not be present in a machine that does not run object implementations, but only clients.

indicate whether the ORB is called or performs an up-call across the interface.

The request paths from a client to an object implementation are shown in Figure 1.3. To request a service from an object, the client must have access to an *object reference* that identifies this object and must know its interface type. The client can perform the request by using either the *dynamic invocation interface* (DII), which allows the dynamic construction of requests at runtime, or a stub generated by the IDL compiler. The former is an interface identical for all ORB implementations and independent of the particular interface to the target object. The latter was generated from the IDL definition of the interface to the target object, and hence is specific to this interface. What distinguishes these two ways of issuing a request is the moment (runtime versus compilation time) the request is known at the client side. From the target object's perspective, dynamic invocation and client stubs have the same semantics: the receiver of a request cannot tell how the request was made.

The object implementation receives the request as an up-call from the object adapter, either through the IDL-generated skeleton or through the dynamic skeleton interface. Requests are normally delivered via IDL-generated skeletons, which are specific to both the object interface and the object adapter. The dynamic skeleton interface addresses the special case of an object implementation that does not have compile-time knowledge of the interface it is implementing[2].

While processing the request, the object implementation may use services from the ORB through the object adapter. It may also directly call the ORB interface to perform certain operations that are common to all objects. Because these operations are useful to both clients and implementations of objects, they are provided by the ORB interface and not by the object adapter, which is unavailable to clients. There are only a few such operations — most of the functionality the ORB offers object implementations is delivered through the object adapter. There may be multiple object adapters, with interfaces that are appropriate for specific kinds of objects. The interface implementor may choose which one to use, based on the needs of a

[2]Strange as it may seem, the dynamic skeleton interface was motivated by a practical problem: interoperability between ORBs through generic bridges. A generic bridge acts as a proxy of objects that live in a foreign ORB environment. It "implements" these objects without having compile-time knowledge of their interfaces.

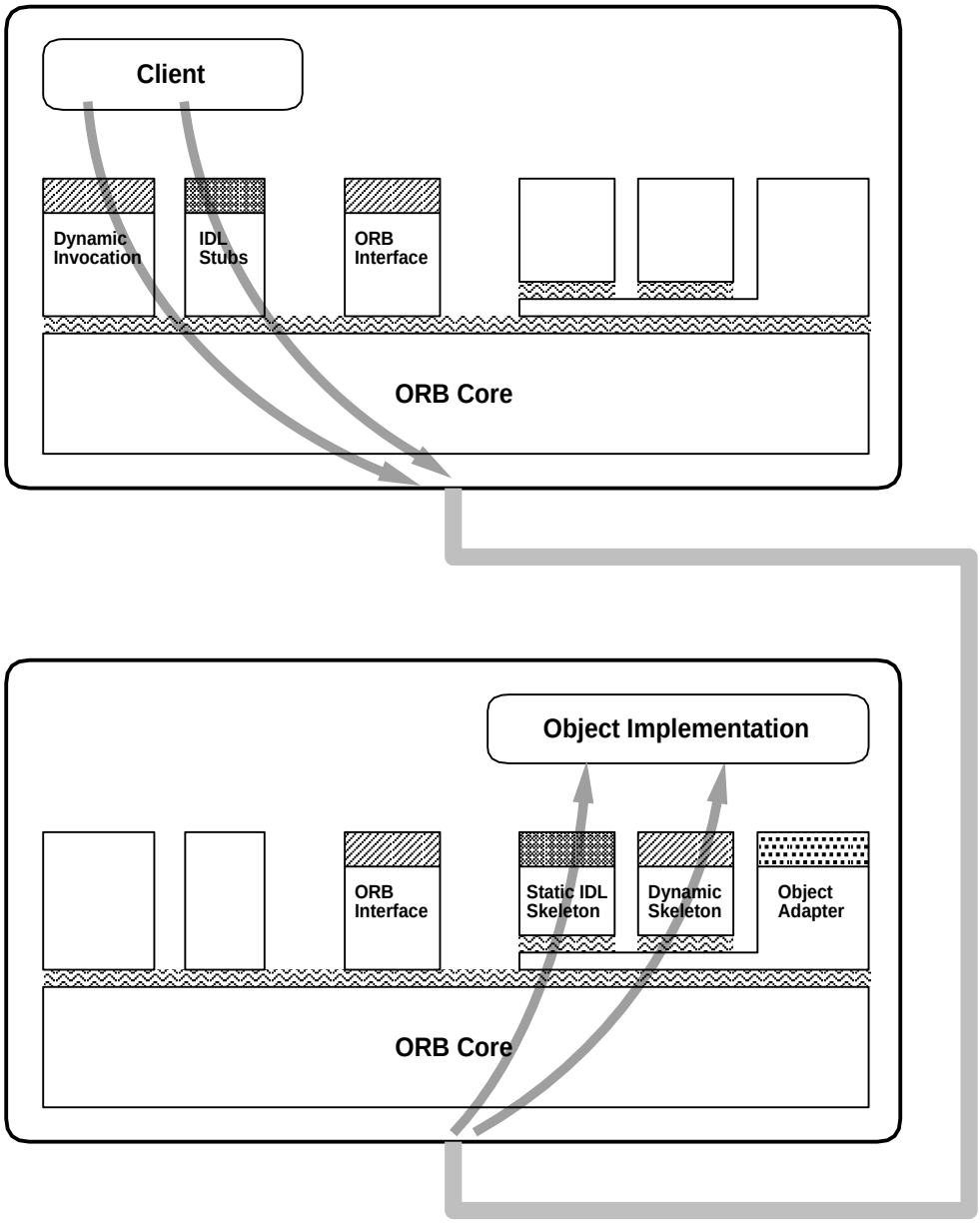



Figure 1.3: Request paths from a client to an object implementation.

particular implementation.

When the object implementation concludes the processing of the request, control and result information is returned to the client. The steps involved in sending a request through an IDL stub are shown in Figure 1.4. The client invokes an operation by calling a stub. The operation arguments are marshaled into a request message, which is sent through the ORB core. These arguments are unmarshaled at the object implementation side, where the request is delivered to the object as an upcall through an IDL-generated skeleton. The results returned by this upcall are marshaled into a reply message, which is sent back to the client through the ORB core. These results are unmarshaled at the client side and returned to the client process.

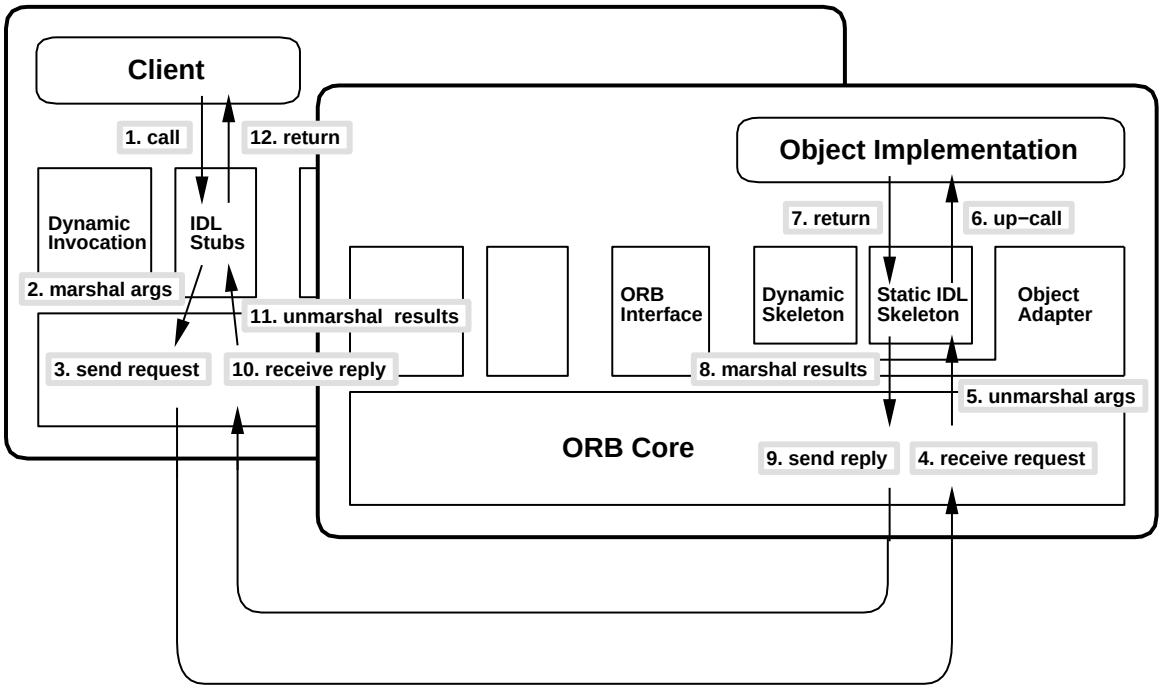



Figure 1.4: A request sent through an IDL stub.

In order to understand how the responsibilities of an ORB are divided among its components, it is useful to list a number of functions of the ORB architecture:

1. Description of object interfaces in a 'language-neutral' way.

2. Generation and interpretation of object references that uniquely identify the objects they reference.

3. Activation and deactivation of object implementations (i.e., bringing up and down *servers*[3] that implement the object interfaces).

4. Activation and deactivation of individual objects (i.e., notifying a server that a particular object should be made available to receive service requests).

5. Transparent transport of service requests and responses. This involves the following tasks:

   (a) marshaling service request arguments into a request message;

   (b) locating the server that should receive this message, based on a reference to the target object;

   (c) delivering the request to the appropriate server;

   (d) unmarshaling request arguments in the server;

   (e) making available to the server the identification of the principal on whose behalf the request is being made;

   (f) locating the target object within the server;

   (g) invoking the corresponding method of this object through the skeleton;

   (h) marshaling results into a reply message;

   (i) delivering the reply to the client;

   (j) unmarshaling results in the client and making them available to the client code;

   (k) converting request and reply messages between data representations used by different machines;

   (l) managing the network connections between clients and servers.

The partition of these responsibilities between IDL, ORB core, and object adapter follows the purposes of these ORB components:

**IDL:** language independence (function 1).

[3]The CORBA specification uses the term *server* for the separately executable entity that implements an object. In a POSIX environment, this would be a process.

**ORB core:** communication infrastructure (tasks 5a–5d, 5h–5l).

**Object adapter:** mediator between the ORB and object implementations (functions 2, 3, and 4; tasks 5e–5g).

## 1.3 The ORB interface

The ORB interface provides operations that are independent of the object adapter, are the same for all ORBs and all object implementations, and can be invoked either by clients of the objects or by implementations themselves. Two IDL interfaces — `ORB` and `Object` — are used to describe such operations. The `Object` interface, inherited by every CORBA object, groups the ones that operate upon object references.

Figure 1.5 presents a partial description of the ORB interface in pseudo-IDL[4], showing only the operations relevant to ORB/ODBMS integration[5]. Note that some of the "ORB interface operations" appear to be on the ORB, while others appear to be on the object reference itself.

The operation `object_to_string` converts an object reference into a string. Such a string may be passed as an input parameter to `string_to_object`, which returns the corresponding object reference. Converting object references to strings is useful because object references are not suitable to be stored in persistent storage or communicated by any means other than operation invocation — they are opaque and their format may differ from ORB to ORB.

Also as a consequence of object reference opaqueness, clients and implementations cannot allocate memory for object references, nor can they check whether or not an object reference actually refers to some object. Copying object references must be done through calls to `duplicate`. When an object reference is no longer needed, its memory should be reclaimed by the use of `release`. Comparisons to `OBJECT_NIL` —

[4]Pseudo-IDL is standard IDL augmented to describe interfaces to CORBA *pseudo-objects*. To its clients, a pseudo-object looks similar to a normal CORBA object; its operations are invoked in the same way as ordinary object operations. Pseudo-objects, however, may be implemented either as normal CORBA objects or as *serverless objects*. Unlike ordinary CORBA objects, serverless objects are not registered with any ORB, and their interfaces do not inherit from `CORBA::Object`. An additional keyword, `pseudo`, is used in pseudo-IDL to distinguish interfaces to pseudo-objects.

[5]Most of the operations omitted from Figure 1.5 are related to the dynamic invocation interface.

```
module CORBA {

    interface InterfaceDef; // from Interface Repository
    interface ImplementationDef; // from Implementation Repository
    interface Object; // an object reference
    ...

    pseudo interface ORB {
        string object_to_string(in Object obj);
        Object string_to_object(in string str);
        ...
    };

    interface Object {
        boolean is_nil();
        Object duplicate();
        void release();
        ImplementationDef get_implementation();
        InterfaceDef get_interface();
        ...
    };
    ...
};
```

Figure 1.5: Partial description of the ORB interface.

a special object reference that denotes no object — are performed through calls to `is_nil`. Note that `duplicate`, `release`, and `is_nil` operate on object references, and not on the corresponding object implementations. Implementations are neither involved in the execution of these operations nor affected by it in any way. In particular, `release` does not delete object implementations.

Given a reference to an object, a description of its interface can be obtained by the use of `get_interface`. This description, which includes type information that may be useful to a program, is returned as an object of the interface repository. Similarly, the `get_implementation` operation on an object reference returns the implementation repository object that describes the implementation of the target object.

## 1.4 The Basic Object Adapter

The Basic Object Adapter has the following responsibilities:

- activation and deactivation of implementations;
- activation and deactivation of individual objects;
- generation and interpretation of object references;
- identification of the principal making a request;
- method invocation through skeletons.

### 1.4.1 Implementation Activation and Object Activation

At the server side, object requests are delivered by the ORB core to the BOA, which in turn invokes the corresponding methods through skeletons. Before calling a skeleton method, the BOA may need to perform two kinds of activation. The first, *implementation activation*, is needed when no implementation of the object interface is currently available to handle the request. The second, *object activation*, when the particular object instance is not available to handle the request.

Implementation activation involves starting the appropriate server, in an operating system-dependent way. Once activated, the implementation may make calls to the BOA. Figure 1.6 presents a description of the `BOA` interface in pseudo-IDL. This interface defines the BOA operations that may be called by implementations. Other interactions between the BOA and object implementations take place by the adapter's initiative: besides activating implementations, the BOA performs up-calls to activate objects and to invoke methods through skeletons.

The BOA supports four policies for implementation activation: *shared server*, *unshared server*, *server-per-method*, and *persistent server* (see Figure 1.8). The first one is likely to be the most commonly used, because it supports automatic server activation in a fashion that uses less machine resources.

#### Shared Server, or Server-per-Class

Multiple active objects of a given interface share the same server under this policy. The BOA interacts with the implementation in the following way (see Figure 1.7):

```
module CORBA {

    interface InterfaceDef; // from Interface Repository
    interface ImplementationDef; // from Implementation Repository
    interface Object; // an object reference
    pseudo interface Principal; // for the authentication service
    typedef sequence<octet,1024> ReferenceData;
    ...

    pseudo interface BOA {

        // implementation activation and deactivation:

        void impl_is_ready(in ImplementationDef impl);
        void deactivate_impl(in ImplementationDef impl);
        void obj_is_ready(in Object obj, in ImplementationDef impl);
        void deactivate_obj(in Object obj);

        // generation and interpretation of object references:

        Object create(
            in ReferenceData id,
            in InterfaceDef intf,
            in ImplementationDef impl,
        );
        ReferenceData get_id(in Object obj);
        void dispose(in Object obj);
        void change_implementation(
            in Object obj,
            in ImplementationDef impl
        );

        // identification of the principal making a request:

        Principal get_principal(in Object obj, in Environment ev);
    };
    ...
};
```

Figure 1.6: The BOA interface.

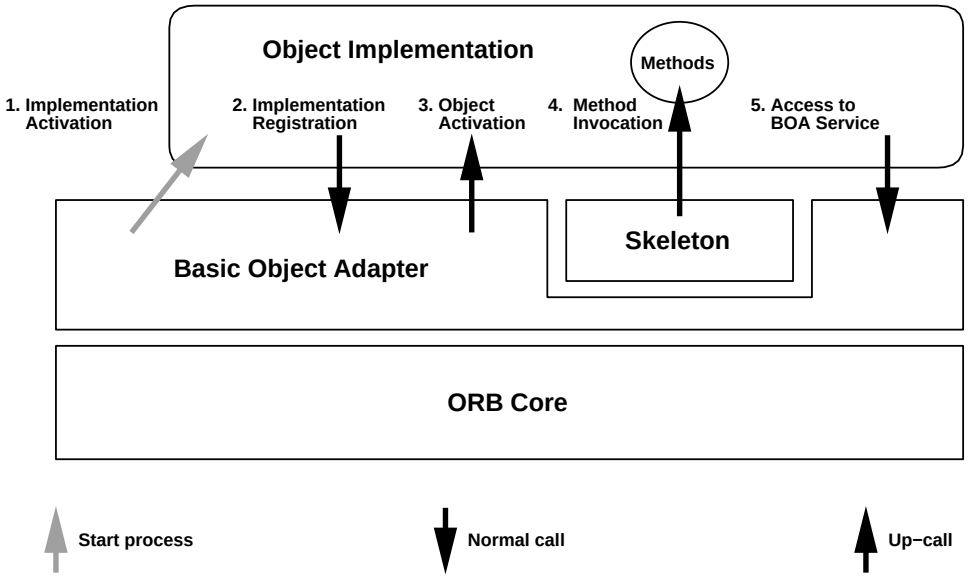



Figure 1.7: BOA operation in a shared server.

1. Implementation activation: at the arrival of the first request to an object implemented by a server, the BOA activates that server.

2. Implementation registration: when the server has initialized itself, it notifies the BOA that it is prepared to handle requests by calling `impl_is_ready`.

3. Object activation: before delivering the first request to a particular object instance, the BOA performs an up-call to the object activation routine of the server. The object remains active as long as its server is active, unless the server calls `deactivate_obj` for it.

4. Method invocation: the BOA delivers requests to object instances by performing up-calls to the appropriate skeleton methods.

5. Access to BOA services: the implementation may call the `BOA` interface to perform operations such as object deactivation, object reference creation, object reference deletion, etc.

The server remains active until it calls `deactivate_impl`.

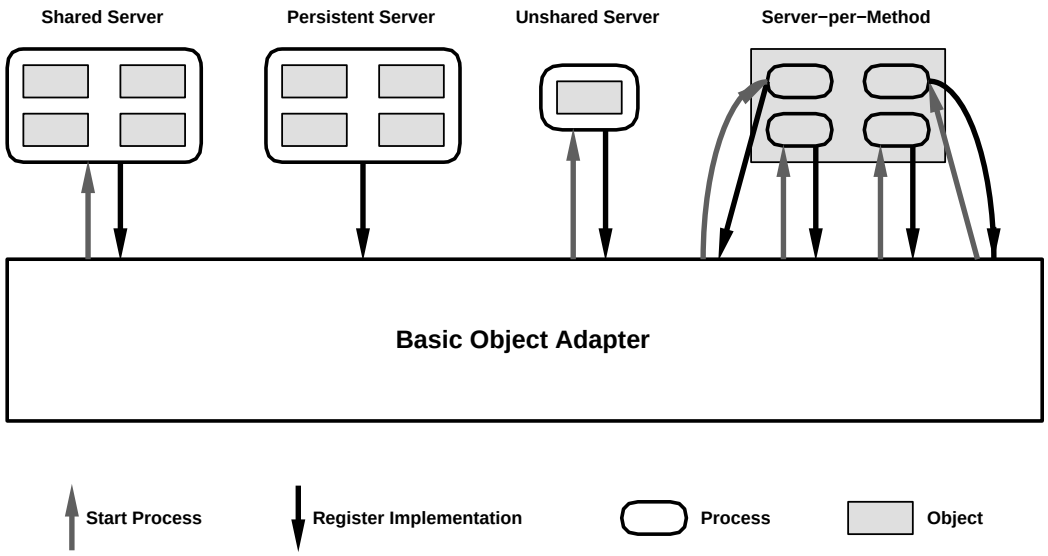



Figure 1.8: Implementation activation policies.

### Unshared Server, or Server-per-Object

Only one object can be active per server under this policy. A separate server is started for each active object instance. The interactions between the BOA and implementations are similar to the ones that take place under the shared server policy, with the following differences:

1. Implementation activation: the BOA activates a new server whenever it receives the first request to an object instance.

2. Implementation registration: when a server has initialized itself, it notifies the BOA that it is prepared to handle requests by calling `obj_is_ready`.

3. Object activation: not needed in this case.

The server remains active until it calls `deactivate_obj`.

### Server-per-Method

A separate server is started for each method invocation; the server exits when the method completes its execution.

**Persistent Server**

This is a variant of the shared server policy, with a single difference: the server is activated by means external to the BOA[6]. After initializing itself, the server calls `impl_is_ready` to notify the adapter that it is available. The interactions between the BOA and the implementation then proceed exactly as in the shared server case.

### 1.4.2 Generation and Interpretation of Object References

An object reference encapsulates all information the ORB needs to identify the same object whenever the reference is used in a request. In addition to this information, an implementation-chosen value, or `id`, is associated with the object reference. Such an `id` should be regarded as a mere component of the object reference, and not as an object identifier that reliably denotes a particular object in a distributed environment — the full object reference has this purpose. The `id` is intended to hold identification information local to an object implementation. Its value is chosen by an implementation, presumably in a way that uniquely identifies an object among the ones the implementation created, and is only meaningful to that implementation.

When an implementation creates an object reference, it specifies an `id` value for the object reference. On subsequent activations of the object, the BOA makes the `id` value available to the implementation, which can interpret the identification information it associated with the object reference and perform the corresponding actions within the object activation routine.

Implementations create new object references by calling to the BOA operation `create`. This operation takes three input parameters: `id`, `intf`, and `impl`. The `id` is the identification information to be associated with the object reference. The `intf` is the object of the interface repository that specifies the interface to the target object. The `impl` is the object of the implementation repository that specifies the implementation of the target object.

Although the object reference itself is opaque and may be different from ORB to ORB, the `id` value is available portably across all ORBs. The BOA operation `get_id`

[6]The usage of the term "persistent server" by the OMG is somewhat misleading, as the CORBA notion of persistent server bears no relationship with the common meaning of persistence. The term "externally activated server" would be preferable.

may be called by implementations to retrieve the `id` associated with an object reference. This is the only information a portable implementation may use to distinguish object references.

If an implementation wishes to delete an object it has created, it should call the BOA operation `dispose` to invalidate its object reference. The implementation is responsible for the deallocation of any other storage areas used by the object.

An implementation may also change the implementation repository object associated with a given object reference. This is accomplished through the BOA operation `change_implementation`. Subsequent requests to the object will be handled by the new implementation.

### Support for Persistent Object References

Although the BOA does not provide persistent object references, it offers mechanisms that can be used to make object references 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 implementation has been deactivated or the system has been restarted" [37].

This can only be achieved with the participation of the implementations, which must keep all object state information in persistent storage. The CORBA specification suggests that implementations use the `id` values to locate object state information in persistent storage — an `id` value might contain the name of a file, or a key for a database that keeps the persistent states of many objects. This suggestion leads to the following scheme (a shared or persistent server is assumed):

1. At object creation time, the implementation chooses object reference `id`s that can later be used to retrieve, from persistent storage, the states of the corresponding objects.

2. When a request to an object that is not active is received, the object activation routine of the implementation is called. The `id` of the target object is passed to this routine, allowing it to retrieve the target object's state from persistent storage. The object activation routine constructs, in virtual memory, an object instance with this state. The incoming request will then be handled by that instance.

Unfortunately the BOA specification left open some important issues, with respect to both object activation and object deactivation. As a consequence of these omissions, non-standardized ORB features are needed to implement the scheme above.

**Object Activation Issues.** The current release of CORBA does not provide a complete specification of the object activation mechanism. All that it says is that the object activation routine is called before the first request is delivered for a particular object, and that the `id` associated with the target object reference is provided to the implementation upon the activation of the object.

How does the BOA know the object activation routine to be called? What are the input parameters to this routine? Does it return a reference to the newly activated object back to the BOA? CORBA left these details to be filled in by ORB implementors. Some ORBs simply do not support object activation at all. Others provide a default activator class that can be refined by the object implementor. Yet others allow the activation routine to be passed as a parameter to an extended `impl_is_ready` operation.

Unfortunately CORBA also left to ORB implementors more significant decisions:

- What about object references appearing as input parameters in a request? If the objects they reference are not active, do they also get activated?
- If an execution of `string_to_object` by a server returns a reference to some local object, does the object get activated?

Some ORB implementations provide object activation mechanisms that give sensible answers to these questions; one such mechanism will be examined in Section 1.6.

**Object Deactivation Issues.** CORBA includes object deactivation among the functions of the BOA, but leaves to the implementation the responsibility of calling `deactivate_obj`. Consider a scenario in which the persistent states of many objects are kept in a large database. Activating these objects means bringing them to an executable context, in which they are ready to handle requests. If they are never deactivated, virtual memory will be exhausted after a large enough number of object activations.

Before it affects the implementation, memory exhaustion may be experienced by the ORB, which certainly maintains in-memory information about active objects. To handle this problem properly, an implementation needs further support from the ORB: a notification that one or more objects should be deactivated, for example. CORBA does not define an object deactivation up-call.

Several ORBs allow the registration of *request and reply handlers* by object implementations. These handlers, which will be discussed in Section 1.7, are non-standardized hooks that may serve several purposes. We will see that they can be used to implement a simple object deactivation discipline.

**To summarize:** ORB features not fully standardized are currently needed to make object references persistent. The standard BOA mechanisms must be extended with

- a fully defined object activation scheme, and
- additional support to object deactivation.

A number of ORB implementations provides such extensions. These issues, as well as other problems in the BOA specification, are being addressed: the OMG, recognizing that server portability requires further standardization of BOA services [30], issued a request for proposals on this subject [41].

### 1.4.3 Identification of the Principal Making a Request

For every method activation or object activation, the BOA makes available to the implementation the identification of the principal on whose behalf the request is being performed. The implementation can get this information by calling `get_principal`[7].

The typical use of `get_principal` is to implement a security mechanism based on access rights. Stronger security mechanisms (*e.g.* message encryption/decryption) may be transparently realized using non-standardized ORB features, such as the request and reply handlers which will be examined in Section 1.7.

[7]The precise meaning of *principal* is operating system-dependent, and hence not specified by CORBA. In a POSIX environment, `get_principal` would return the username of the process that made the request.

## 1.5 IDL to C++ Mapping Issues

Rather than describing the IDL to C++ mapping specification ([37], chapters 15–18), we discuss some issues not specified by this standard, but relevant to ORB/ODBMS integration.

In a C++ environment, both client stubs and implementation skeletons are C++ classes. For each interface, the IDL compiler generates an *interface class*, with stub member functions to be called by clients, and a *skeleton class* to be used by the interface implementation. Interface classes provide the means through which clients perform requests. Skeleton classes, as the term "skeleton" suggests, do not implement interfaces. They merely act as a path from the BOA to the actual *implementation classes* supplied by the interface implementor, allowing delivery of remote requests to implementation objects[8] through up-calls to member functions of these objects.

An ORB implementation must give answers to the following questions:

1. How does the IDL compiler map interface inheritance to C++ constructs?
2. How does the interface implementor indicate that an implementation class realizes a given IDL interface?

Both questions have natural answers, which current ORB implementations already provide. Such answers are also mentioned — but not mandated — by the IDL to C++ mapping specification.

### 1.5.1 The Natural Mapping of Interface Inheritance

Because every CORBA object inherits from `CORBA::Object`, interface inheritance occurs even when an IDL interface definition does not specify a base interface. Consider the example of IDL interface in in Figure 1.9, extracted from chapter 18 of [37]. Figure 1.10 shows the approximate form of the interface class definition generated by an IDL compiler for this interface.

Note that the interface class in Figure 1.10 has `CORBA::Object`, the base of the interface class hierarchy, as its virtual base class. Although the IDL to C++ mapping

[8]Terminology gets a bit confusing here. An *implementation object* is an instance of an implementation class. An *object implementation* is a server that implements an object's interface. The latter is also called *interface implementation*, or simply *implementation*.

```
// IDL
interface A {
    short op1();
    void op2(in long l);
};
```

Figure 1.9: IDL definition of interface A.

```
// C++
class A : public virtual CORBA::Object {
public:
    virtual CORBA::Short op1();
    virtual void op2(CORBA::Long l);
    ...
};
```

Figure 1.10: IDL-generated interface class for interface A.

does not require such inheritance to be virtual, this is actually the case in most if not all current ORB implementations. Since IDL supports multiple inheritance of interfaces, inheritance relationships between IDL interfaces are naturally mapped to C++ virtual inheritance relationships between the corresponding interface classes. The natural mapping of interface inheritance to C++ is illustrated by Figures 1.11 and 1.12.

```
// IDL
interface B : A { ... };
interface C : A { ... };
interface D : B, C { ... };
```

Figure 1.11: IDL interfaces B, C, and D.

Due to the use of virtual inheritance, a single sub-object of class `A` is shared by parts `B` and `C` of class `D`. Moreover, a single sub-object of class `CORBA::Object` is shared among class `D`, its parts `B`, `C`, and their common sub-object of class `A`. The

```
// C++
class B : public virtual A { ...  };
class C : public virtual A { ...  };
class D : public virtual B, C { ...  };
```

Figure 1.12: IDL-generated interface classes for interfaces B, C, and D.

second remark is the strongest reason to translate IDL interface inheritance into C++ virtual inheritance. While it might be desirable, under very special circumstances, to have separate sub-objects of class `A` in parts `B` and `C` of class `D`, it is out of question that a single sub-object of class `CORBA::Object` should be shared by class `D` and its parts[9].

Although the IDL to C++ mapping specification does not mandate any approach to the C++ translation of interface inheritance, and even states that IDL interface inheritance does not necessarily imply an inheritance relationship between the corresponding interface classes, most if not all current ORB implementations use the natural mapping we just described.

### 1.5.2 Implementing Object Interfaces in C++

Our focus now is in the so called *server side* mapping, which deals with specific issues whose interest is restricted to interface implementors using C++.

The interface implementor must have a way to indicate that an implementation class realizes a given IDL interface. The natural way is through some relationship between the implementation class and the skeleton class obtained from the interface. To avoid constraining ORB implementations, the standard IDL to C++ mapping does not specify how these classes are related. It describes two possible approaches (Figure 1.13) — one based on C++ inheritance, the other using delegation — but does not require ORB implementors to follow any of them. Most if not all current ORB implementations support the inheritance approach; some of them support both approaches and leaves the choice to the interface implementor. Clients are not affected

[9]Since class `CORBA::Object` provides access to the ORB communication infrastructure, it does not make any sense for a CORBA object to have more than one sub-object of this class.

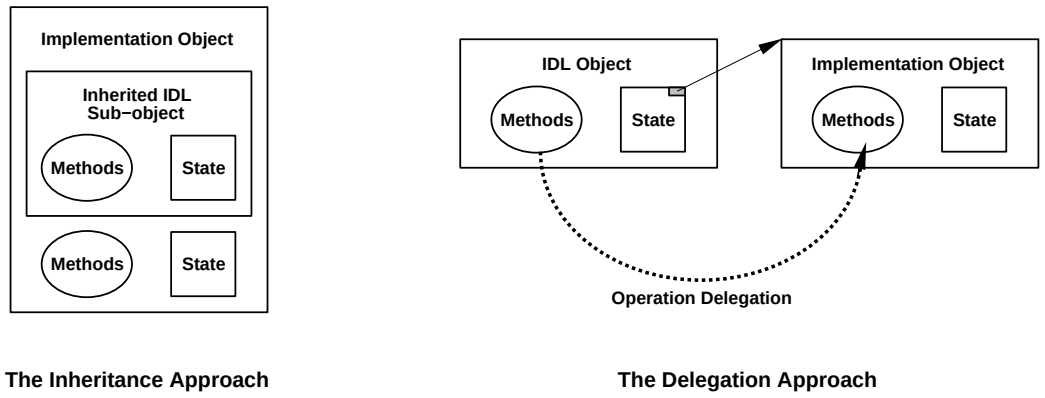



Figure 1.13: Interface implementation approaches.

```
// C++
class skeleton_A : public A {
public:
    ...  // ORB-dependent member functions go here
    virtual CORBA::Short op1();
    virtual void op2(CORBA::Long l);
    ...
};
```

Figure 1.14: IDL-generated skeleton class for interface A.

by the approach a server uses to relate implementation and skeleton classes.

### The Inheritance Approach

In this approach, implementation classes are derived from the skeleton classes generated by the IDL compiler. The skeleton class corresponding to an interface declares a virtual member function for each operation of the interface. These member functions are redefined by the implementation class, which actually provides their executable bodies. The BOA invokes the implementation methods via up-calls to the virtual functions of the skeleton class.

As an example, consider the IDL interface shown in Figure 1.9, and its IDL-generated interface class definition, shown in Figure 1.10. The skeleton class for this interface looks like the one in Figure 1.14.

```
// C++
class A_implementation :  public skeleton_A {
public:
    CORBA::Short op1();
    void op2(CORBA::Long l);
    ...
};
```

Figure 1.15: Implementation class for interface A (inheritance approach).

Figure 1.15 shows how an implementation class for interface A would be declared. The interface implementor must derive the implementation class from the skeleton class, redefine all member functions that correspond to interface operations, and provide the implementations of these operations. Note the inheritance ("is a") relationship that `A_implementation` holds with `skeleton_A`.

### The Delegation Approach

In this approach, the IDL compiler generates a special form of skeleton class, called a *tie*. This class is a wrapper that holds a reference to an implementation object, and delegates up-calls to this object.

A tie class has a data member that refers to an instance of the implementation class, and defines a member function for each operation of the interface. These member functions are typically one-line functions that use the reference to the implementation class to call the corresponding implementation methods.

As an example, consider again the IDL interface shown in Figure 1.9, and its IDL-generated interface class definition, shown in Figure 1.10. The tie class for this interface looks like the one in Figure 1.16.

The tie is a class template; this scheme provides type safety. In a C++ environment without support for templates, the IDL compiler might generate a tie macro instead. The argument to the tie template is the name of an implementation class. Implementation classes are not derived from any IDL-generated class; the interface implementor only needs to define a class whose member functions implement all the interface operations, such as the one in Figure 1.17.

```
// C++
template <class Impl>
class tie_A : public A {
private:
    Impl& ref;
public:
    ...  // ORB-dependent member functions go here
    tie_A(Impl& impl_obj) :  ref(impl_obj) {}
    CORBA::Short op1() { return ref.op1(); }
    void op2(CORBA::Long l) { ref.op2(l); }
    ...
};
```

Figure 1.16: IDL-generated tie class for interface A.

Implementation objects are "tied" to IDL interfaces at runtime. For example, the execution of

```
tie_A<A_impl> a_obj = tie_A<A_impl>(a_impl);
```

(where `a_impl` is an object of class `A_impl`) creates a "tie object" `a_obj`, which satisfies interface A and is has `a_impl` as its implementation object. Note the delegation ("has a") relationship between the tie class `tie_A<A_impl>` and the implementation class `A_impl`.

```
// C++
class A_impl {
public:
    virtual CORBA::Short op1();
    virtual void op2(CORBA::Long l);
    ...
};
```

Figure 1.17: Implementation class for interface A (delegation approach).

### Comparison of the Inheritance and Delegation Approaches

In the delegation approach, implementation classes are not constrained by the inheritance hierarchy of the corresponding interface classes, which typically reflects the inheritance hierarchy of IDL interfaces. Although it is common to design implementation classes that mimic the inheritance hierarchy of their IDL counterparts, this practice is inconvenient or impossible in many cases — when implementing IDL interfaces with existing legacy code, for example.

The inheritance approach, on the other hand, imposes the inheritance hierarchy of interface classes to the interface implementor. If this imposition is unacceptable, the implementor must resort to multiple inheritance in order to circumvent it. Consider a situation in which a programmer wants to provide IDL interfaces to legacy classes whose inheritance hierarchy is different from the one of the desired interfaces. The programmer is forced to define, for each IDL interface, a "front-end" implementation class derived from both the IDL-generated interface class and the legacy class that implements the interface.

In spite of the significance of the remarks above to an interface implementor, the ones that follow are yet more relevant to ORB/ODBMS integration.

The distinction between IDL objects and their implementation objects is much weaker in the inheritance approach. Under this approach, the "IDL part" of an implementation object is an inherited sub-object — its skeleton part, derived from an interface class, which in turn inherits from `CORBA::Object`. Whenever an implementation object is created, its IDL part will also be instantiated. It is not possible for an implementation object to outlive its IDL part.

The delegation approach, by contrast, provides a clear separation between IDL objects — tie objects — and their implementations. An implementation object does not hold any interface class instance as an inherited sub-object, nor does it inherit any sub-object of class `CORBA::Object`. Implementation objects are instantiated before the creation of the corresponding tie objects, and may also outlive these tie objects.

## 1.6 Object Activation in Orbix

Iona Technologies' Orbix [18] is a commercial ORB product that fully supports object activation. It also supports both the inheritance and the delegation approach to interface implementation, and allows the installation of request and reply handlers (Section 1.7) by object implementations. Here we describe the design decisions that Orbix implementors have made with respect to object activation [17].

Orbix introduces its own terminology for object activation concepts. It refers to the events that trigger the object activation routine as *object faults*. Object activation is called *loading*, and is performed by *loader* objects. To define a loader class, the interface implementor defines a class that inherit from the class `CORBA::LoaderClass` provided by Orbix. A server installs a loader at runtime, by creating an instance of a class derived from `CORBA::LoaderClass`. The loader's object activation routine is the virtual member function `load` of `CORBA::LoaderClass`; interface implementors may redefine this function in their loader classes.

In Orbix, the object activation routine is called whenever a reference to a local object that is not currently active enters the server's address space. An object reference can enter an address space in one of the following ways:

1. as the target of a request received by a server;
2. as an `in` parameter of a request received by a server;
3. as an output (`out` or `inout`) parameter or as the return value of a request made by a client (which may be a server requesting a service from another server);
4. as result of the conversion of a string into object reference, either by the ORB operation `string_to_object`, or by the Orbix-specific function `_bind`.

The input parameters to the object activation routine are:

- the name of the object's interface;
- the `id` (*marker*, in Orbix terminology) associated with the object reference;
- a boolean value that tells whether the routine was triggered by the conversion of a string to object reference or not.

Note that an object reference does not enter an address space in its actual form. Either it arrives in marshaled form, or will be generated from a string. When the object activation routine is called, the object identified by the incoming reference does not exist in the address space of the server; the activation routine reconstructs this object in the server's address space and returns an actual reference to it. More precisely, the object activation routine returns a reference to the `CORBA::Object` part of the reconstructed object. Further conversion of this reference into one that refers to a more specific interface[10] may be necessary; skeleton code performs such conversion in the case of references that entered the address space in marshaled form.

## 1.7 Request and Reply Handlers

This facility allows user-defined procedures to be triggered by incoming or outgoing messages. Such procedures, or *handlers*, are executed in addition to the normal processing of incoming and outgoing messages. Several ORB implementations provide request and reply handlers in a variety of ways.

ORBeline [46], for example, offers request and reply handlers as a BOA extension available only at the server side. The request handler (*pre-method*, in ORBeline's terminology) is invoked whenever a client performs a request to the server. The reply handler (*post-method*) is invoked after the requested operation completes and before the reply is sent to the client. Both handlers receive, as input parameters, the identification of the principal making the request, the name of the interface of the target object, and a reference to this object.

Orbix offers such a facility at both the server and the client side; its handlers (*filters*, in Orbix terminology) cover all communication events of interest. Message handlers may be associated with message arrival and with message departure, and may be triggered either before or after the message is marshaled. Four types of message handlers may be defined at the server side:

- incoming request pre- and post-marshal;
- outgoing reply pre- and post-marshal;

[10] Although C++ does not allow downcasting from a virtual base class, the IDL to C++ mapping provides the `_narrow` function to support downcasting of interface types.

The corresponding four handlers at the client side are:

- outgoing request pre- and post-marshal;
- incoming reply pre- and post-marshal.

All these handlers receive, as an input parameter, a reference to a `Request` instance that describes the current request[11]. They can use this reference to obtain detailed information about the request, including target object and operation name.

Because request and reply handlers are needed by services such as security, transaction management, and replication, a similar facility was recently standardized by the OMG: the Security Service Specification [2] introduced *request level interceptors* and *message level interceptors* as an extension to the ORB core. These interceptors provide the functionality of the request and reply handlers examined here.

## 1.8 Additional Information

This chapter covered aspects of the ORB architecture that are relevant to our main subject. We now mention briefly a number of topics left out.

Important additions to the OMG standards happened in 1995. The revision 2.0 of the CORBA specification solved the problem of interoperability between ORBs provided by different vendors. It defined *inter-ORB bridges* and specified the Internet Inter-ORB Protocol (IIOP), a protocol that runs directly over TCP/IP. Moreover, significant progress was made on the definition of services provided not by the ORB, but by external components, in a modular fashion.

The ORB is just the backbone of the OMG architecture. By itself, it does not perform any of the higher-level services one would expect in a distributed object system. These services are provided by "system objects", components with IDL interfaces that can be introduced into a CORBA environment to satisfy specific needs. A growing set of *object services* is being defined by the OMG. The following ones were specified in [39]:

- Naming Service,

[11] A `Request` is a CORBA pseudo-object, like `ORB` and `BOA`. `Request`s provide the primary support for the dynamic invocation interface.

- Event Service,
- Persistent Object Service,
- Lifecycle Services,
- Concurrency Control Service,
- Externalization Service,
- Relationship Service,
- Transaction Service.

Implementations of some of these services are starting to appear, and more are expected, still during 1996. Additional object services were standardized in 1995:

- Query Service,
- Licensing Service,
- Properties Service,
- Security Service,
- Time Service.

Furthermore, standardization work is in progress on the following ones:

- Trader Service,
- Collections Service,
- Change Management Service.

This set of object services addresses essential and general needs. Higher level or specific needs are addressed by *object facilities* with interfaces closer to the application level. Two sets of *object facilities* are being specified: one grouping facilities independent of the application domain (*horizontal facilities*), other with facilities targeted at specific application domains (*vertical facilities*).

The OMG specifications (e.g., [39], [38], [16], [2]) are the definitive source of information on these services and facilities. For an introductory overview, covering most aspects of the OMG architecture, we recommend [44].

# Chapter 2

# Object Database Management Systems

A comprehensive discussion of ODBMSs is beyond the scope of this dissertation. This chapter presents an overview of ODBMS characteristics, with emphasis on aspects relevant to ORB/ODBMS integration. The reader is referred to [62] for an excellent introduction to the field of ODBMSs, to [6] for a more extensive treatment, to [32] for a good survey of existing systems, and to [7] for an effort to put forward a set of standards for object databases. This standardization effort is being undertaken by the Object Database Management Group (ODMG), a group of ODBMS vendors who have committed to make their products compliant with the proposed standard.

Most ODBMSs extend an existing object-oriented programming language to incorporate database functionality. Research prototypes of such systems started to appear in the late 1980s, those include ORION [23], $O_2$ [9], ODE [1], and E [47] [48]. Current commercially available ODBMSs include ObjectStore [29], $O_2$ [8], GemStone [4], Objectivity/DB [42], ONTOS [43], and Versant [57].

Database application writers using a data manipulation language (typically SQL) embedded in a host programming language have long been burdened by the *impedance mismatch* [62] between the data manipulation language of the database and the general-purpose programming language in which the rest of the application is written. ODBMSs solve this problem by the integration of the programming and database environments. A database application can then be written in a single and computationally complete language.

The benefits of the object-oriented paradigm come as a consequence of the use of an object-oriented language. Object identity [21] can be used as an alternative to the primary keys of relational databases, allowing relationships between objects to be represented more efficiently than in relational DBMSs (by inter-object references), and with a syntax more convenient than relational joins. User-defined types, data encapsulation, and type inheritance are available in the database environment. Objects with complex state (i.e., objects whose attributes are not restricted to primitive types) can be stored in the database.

The vast majority of ODBMSs add database functionality to C++ [10], probably because of the growing popularity of this language. Henceforth we will focus on these systems.

## 2.1 Persistence

In standard C++, objects are allocated from *transient memory* — the virtual memory managed by the operating system — and their lifetime is either *coterminous with procedure* (function arguments and automatic variables) or *coterminous with process* (static objects and heap-allocated objects). ODBMSs extend C++ to support a third object lifetime, *coterminous with database*, for objects allocated out of *persistent memory* (Figure 2.1).

Persistent memory can be either an ODBMS-managed heap in secondary storage,

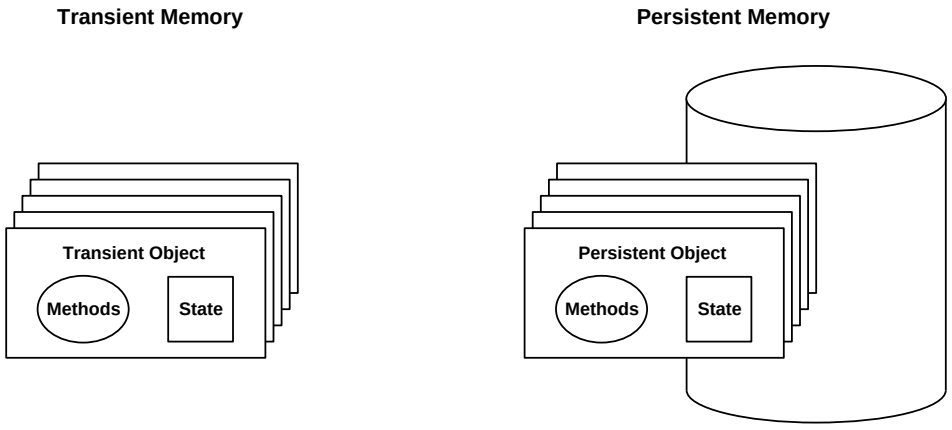



Figure 2.1: Transient memory and persistent memory.

or a collection of storage pages, segments, or clusters managed by the ODBMS. How does the programmer specify that an object's lifetime is coterminous with database? Most ODBMSs fall in one of the following cases:

**Persistence independent of type.** Persistence of an object is explicitly indicated when the object is instantiated. This approach typically uses heap-style allocation. The ODBMS provides an overloaded form of the operator `new` that creates objects in persistent memory. Additional arguments to `new` might specify a database, segment, or cluster.

**Persistence by type.** The system distinguishes persistent types from transient types. Objects are created in transient memory or in persistent memory, depending on their type. Some ODBMSs extend C++ by including a persistence specifier to be used in persistent class declarations; others require that all persistent classes inherit from a "persistent root" class.

The second approach incurs a conceptual flaw that makes it less convenient to the programmer: since persistence is naturally orthogonal to typing, one should be able to specify persistence of an object regardless of its type.

The ODMG specification adopts a mix of these approaches. All persistent objects must inherit from an ODBMS-defined class `d_Object`. Classes derived from `d_Object` are *persistent capable*: by itself, inheritance from `d_Object` does not make instances of these classes persistent. Instances of a persistent capable class can be either persistent or transient. For such classes, an overloaded form of operator `new` is used to specify object persistence at instance creation time.

## 2.2 Collections and Queries

Every ODBMS offers some facility to create and operate upon objects that group other objects, all of the same type, in homogeneous collections. This typically takes the form of a C++ library that defines a hierarchy of collection class templates. The base class of this hierarchy is `d_Collection<T>`, where `T` is any primitive or user-defined type. `d_Set<T>`, `d_Bag<T>`, `d_List<T>`, and `d_Varray<T>` are defined as

subclasses of `d_Collection<T>`[1]

Despite the similarity between the collection hierarchies provided by various systems, ODBMSs vary wildly on how iterations and queries over collections are performed. All systems support at least some way of iterating over all objects in a collection. In most systems (and in the ODMG specification) the iteration facility appears as a C++ library that defines an iterator class template[2]. In others, it appears as language extensions, usually somewhat similar in syntax to the `for` statement of C++, but with declarative (nonprocedural) semantics. These language extensions are a two-edge sword. Their positive side is programming convenience, plus the optimization opportunities allowed by declarative semantics. Their negative side is lack of application portability among ODBMSs.

Current ODBMSs vary even more with respect to embedded queries. In some, the only way to perform a query is through iteration over all objects in the collection being queried. In others, iterations may be restricted to objects that satisfy a logical constraint. Still others offer a true facility for embedded queries. Such a facility performs associative lookup to retrieve, from a given collection, the subcollection of objects satisfying a logical constraint. It may appear as a query class library, or in the form of language extensions with declarative semantics. Almost all ODBMSs, however, still lack the full join capability found in relational DBMSs[3]. Their queries are restricted to *semijoins*, which filter out the elements of a collection that do not satisfy a logical constraint, but cannot create new types by joining together attributes from existing types.

*Ad hoc* facilities for interactive queries are currently absent from almost all ODBMSs[4]. This fact, along with the lack of a standard for a declarative "object query language" and with the weaknesses of the embedded query facilities of existing

[1]These are the collection class names the ODMG specified in [7]. ODMG-defined global names have the prefix "`d_`". Most ODBMS products are still not compliant with the ODMG standard, so the actual names are vendor-dependent.

[2]Instead of a single iterator class template, there may be a hierarchy of iterator class templates whose inheritance graph parallels the one of the collection class hierarchy, as in ObjectStore [29].

[3]The notable exception is $O_2$ [8], which has an SQL-like query facility with full join capability.

[4]Again the exception is $O_2$ [8], whose query facility can be used both for interactive queries or as a function callable from C++.

ODBMSs, is regarded as one of the major drawbacks of these systems. The ODMG addressed this problems by including the definition of an object query language in its specification.

## 2.3 Transactions

Full-fledged ODBMSs provide a transaction facility that supports atomic operations, enforces data consistency in the presence of concurrent accesses, and guarantees data integrity in spite of hardware or software failures. Such facility is analogous to the ones found in traditional database systems.

All accesses to persistent objects must be performed within a transaction. Transactions must be explicitly started and committed (or aborted) by the programmer. A transaction ends when a programmed commit or abort operation is executed, or in the event of a system-generated abort. Some ODBMSs support nested transactions, thus providing a richer transaction model than traditional DBMSs.

The transaction facility is typically accessed through a class `d_Transaction`, which defines the operations `begin`, `commit`, and `abort`. The `begin` operation may take arguments to specify the transaction type (*read-only* or *update*) and the concurrency control mechanism to be used (*pessimistic* versus *optimistic*).

The utilization of object databases is currently stronger in new application areas such as CAD/CAM, CASE, scientific and medical applications, and geographic information systems[5]. In these areas the probability of concurrency conflicts tends to be smaller than in traditional business applications, and the optimistic approach to concurrency control is likely to perform better than the pessimistic one. Even so, the majority of ODBMSs still support only the traditional pessimistic (i.e., lock-based) mechanism.

## 2.4 References to Persistent Objects

Most ODBMSs introduce a reference class template `d_Ref<T>`, whose instances are references to persistent objects of type `T`. Objects of class `d_Ref<T>` are called *database*

[5]These are the application areas that motivated the development of ODBMSs.

*object references*, or simply *ODBMS references*. Overloading of the operators "`->`" and unary "`*`" by class `d_Ref<T>` allows ODBMS references to be dereferenced like `T*`s.

ODBMS references can be used either to point to persistent objects from transient memory, or to express database relationships. Every ODBMS places some restriction on the use of the C++ data types `T*` and `T&` to refer to persistent objects. A common constraint, present in the object database standard proposed by the ODMG, states that C++ pointers or references held by persistent objects are only meaningful during the execution of a transaction; they are invalidated when the outermost nested transaction commits. This restriction effectively disallows the use of C++ pointers or references to express relationships between persistent objects. Some ODBMSs, notably ObjectStore [29], follow a much more liberal approach: C++ pointers or references between persistent objects are allowed in a database and remain valid even when no transaction is being executed. C++ pointers or references from transient memory to persistent memory, however, are invalidated at the end of the outermost transaction in execution. Under this approach, database relationships can be realized either by database references or by more efficient C++ pointers/references. Although copies of these pointers/references may be retained in transient memory after the outermost transaction commits, the application programmer must make sure that no such copies are ever used within subsequent transactions.

Every ODBMS reference reflects the identity of the persistent object it references; database object references are the C++ counterparts of the object identifiers (OIDs) through which the concept of object identity is realized in an object database. An ODBMS object reference is similar to a CORBA object reference, in that they both identify an object and can be used to invoke its methods. This similarity goes even further in some ODBMSs, which support conversion of their object references to strings and vice-versa, just like ORBs do. Conversion of ODBMS references to strings allows such references to be passed between processes and/or persistently stored outside the databases in which the referenced objects live, thus facilitating the construction of heterogeneous object databases and the integration of the ODBMS with other software systems.

## 2.5 Client/Server ODBMSs

Commercially available ODBMSs typically have client/server architectures: one or more *database servers* manage their local storage devices and provide networked *database clients* with access to the objects persistently stored in the databases (Figure 2.2). ODBMS references between persistent objects kept by different servers effectively allow the construction of distributed object databases.

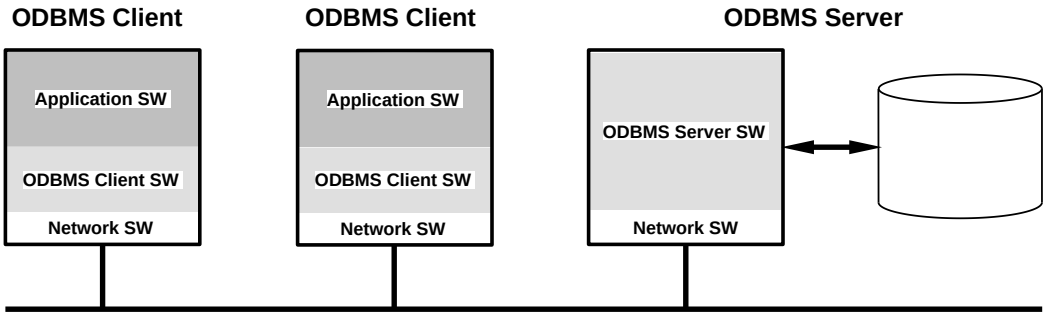



Figure 2.2: Client/server ODBMS.

### 2.5.1 Comparison with CORBA

A client/server ODBMS supports object distribution, but not in the same way ORBs do. CORBA is an "operation shipping" architecture; client/server ODBMSs have "data shipping" architectures (Figure 2.3). While ORBs support the transmission of service requests and responses across a network, client/server ODBMSs support the transmission of object data across the network. In an ORB environment, CORBA clients have access to the operations defined on the objects implemented by CORBA servers. In a client/server object database environment, database clients have access to the persistent states of the objects held by database servers.

Client/server coupling is much looser in CORBA: all that a CORBA client needs to know are the IDL interfaces to the objects it accesses. Database clients, on the other hand, are usually linked to database libraries supplied by the ODBMS vendor, and must have precise knowledge of how persistent objects are logically stored in the database (*database schema*).

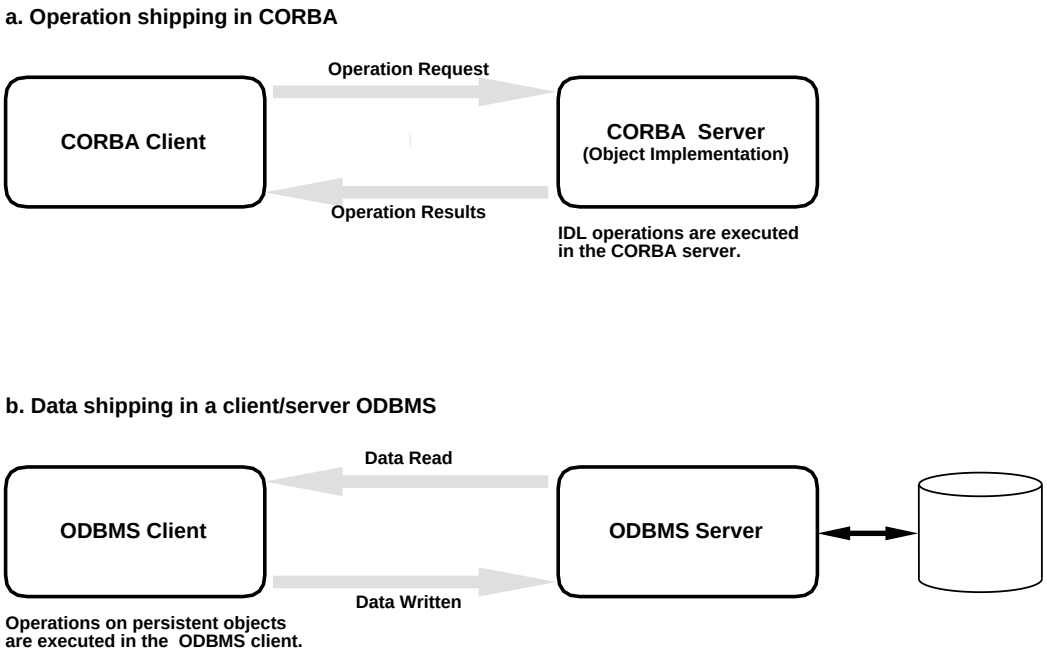



Figure 2.3: Operation shipping vs. data shipping.

## 2.6 ODBMS Implementation Issues

ODBMSs typically make objects persistent through a *memory image* approach. To reduce the need for conversions, the memory representation of persistent objects is designed to follow closely the one used by the programming language in which object database applications are written (C++, in the majority of cases). Representing objects in persistent storage in the same way as in virtual memory is a good match to the *page server* architecture[6] employed by some client/server ODBMSs.

A memory image approach implies a direct relationship between the in-memory size of a persistent object and the database space used by the same object. Any data fields in an object's memory representation — even the ones that may not be actually stored in the database, such as the hidden `vbase` and `vtbl` pointers within C++ objects — affect on the database space taken by that object.

Such an approach, however, does not imply that the external representation of

[6]In this architecture, a database server is actually a page server. Databases are organized as collections of pages, all with the same size, and database servers perform data shipping on a page basis.

a persistent object is exactly identical to the memory representation of the object. While the overall formats are the same, some conversions may still be necessary. The external representation of a database reference is the OID of the object it references. The memory representation of such reference may take another form: for performance reasons, many ODBMSs use the address of the target object in virtual memory, instead of its OID. In-memory conversion from OIDs to virtual memory locations is called *pointer swizzling*. Swizzled pointers are virtual memory addresses, unswizzled pointers are OIDs. Several techniques have been used to translate the external representation of an ODBMS reference into the actual memory location of the object it references. The most common ones are:

1. *No swizzling.* Whenever the reference is followed, the OID contained in it is used to lookup the memory address of the target object, and to retrieve it from the database if it is not in memory yet. Such lookup involves a search in a table maintained in memory by the ODBMS, a procedure relatively time-consuming. The purpose of pointer swizzling is to avoid incurring this cost at every traversal of an ODBMS reference.

2. *Swizzling at first traversal.* The reference is swizzled the first time it is followed. Subsequent traversals will not perform a table lookup, but only a check to verify if the reference is already swizzled or not.

3. *Swizzling at page fault time.* This technique uses the hardware address translation mechanism to map database pages into virtual memory pages. When a virtual memory fault occurs, all ODBMS references contained in the faulting page are swizzled. The traversal of an ODBMS reference causes a page fault if the database page in which the target object resides is not yet mapped to virtual memory. Otherwise, following an ODBMS reference is as fast as dereferencing a normal pointer. When the reference is followed, no software checks are performed to verify whether it is swizzled or not.

None of these techniques is an obvious winner; the best one for a particular application depends on the characteristics of the application. Technique 2 avoids repeated table lookups, but must keep track of the pointers that have been swizzled in order to unswizzle them when the objects they point to leave memory. Technique 3 avoids

performing any software checks, but incurs the cost of swizzling pointers that may never be followed. The reader is referred to [33] for an analysis of the costs of swizzling, to [59] and [50] for a detailed description — and also a defense — of pointer swizzling at page fault time, and to [58] for a recent study of the relative performance of the three techniques above.

With ORB/ODBMS integration in mind, it is important to make a distinction between the first two techniques and the third one.

Techniques 1 and 2 are software-based, in the sense that they rely on the execution of an ODBMS routine whenever a database object reference is followed. We refer to both as *smart pointer* approaches, because their typical realization uses overloading of the C++ operators "`->`" and unary "`*`" — the well-known "smart pointer" scheme [10].

Technique 3, by contrast, is a *virtual memory* approach. It takes advantage of the virtual memory hardware, both to map persistent memory to transient (virtual) memory and to avoid software checks at pointer traversal time.

The ODBMS design decision between smart pointer approaches and virtual memory approaches is one of the most controversial subjects within the object database community. The majority of ODBMSs use smart pointer approaches. Almost all commercially available systems fall in this case; examples include Objectivity/DB and $O_2$.

Only one commercial system — ObjectStore — employs a virtual memory approach. The idea of applying virtual memory techniques to ODBMSs was introduced by ObjectStore. Its implementation, which differs in some significant aspects from the technique described in [59], is briefly outlined in [29]. Research prototypes that use virtual memory approaches include Texas [50], Cricket [49], and QuickStore [58]. These research systems, however, are persistent object managers that do not provide full ODBMS functionality.

Although currently represented by a single commercial system, virtual memory approaches have great significance from a practical standpoint, because ObjectStore holds a large fraction of the ODBMS market. Since a huge virtual address space reduces — and may even eliminate — the need for pointer swizzling, it is likely that 64-bit architectures will make virtual memory techniques yet more attractive. An example of this trend is the virtual memory-based ODBMS research prototype,

targeted at 64-bit architectures, currently under development at the IBM Thomas J. Watson Research Center [31].

## 2.7 ODBMS Limitations

Current object database systems lack several features commonly offered by relational systems. The most serious limitation of existing ODBMSs — the lack of a standardized and non-procedural query language, with automatic query optimization and full support for and for interactive *ad hoc* queries — was already mentioned in Section 2.2. Other limitations include:

**Single-language environment.** Solving the impedance mismatch problem (see page 39) by the integration of the programming and database environments imposes a particular programming language — the one to which the ODBMS adds database functionality — to application writers. Although some ODBMSs give the programmer a choice between C++ and Smalltalk, this is still far from the variety of language options provided by relational systems, which allow SQL to be embedded into a number of programming languages.

**No support for views.** Relational DBMSs support views, virtual tables obtained from the actually stored ones through select and/or join operations. Views are used as dynamic windows into the database, both for data independence (a change in the database schema will not affect users, if they are still given the same views to the database) and for authorization purposes (views can be defined to present only data items that certain users are allowed to read or update). No ODBMS currently supports a mechanism equivalent to relational views.

**Coarse or absent authorization mechanisms.** Relational systems support authorization: users can specify who is allowed to read or update the tables or views they created. Most ODBMS do not support authorization, or provide it in a coarse fashion — all or nothing access, either at the database level or at the segment level. For efficiency, some ODBMSs implement the memory image approach giving database clients access to an entire database, or to database

segments, through the clients' address spaces. This requires a high level of trust on database clients.

**No support for dynamic schema changes.** Relational DBMSs allow dynamic changes to the database schema: new columns may be added to a table, tables may be dropped, and under certain circumstances columns can be dropped from a table. In most existing ODBMSs, changes to the database schema cannot be performed dynamically.

The reader is referred to Won Kim's article [28] for further discussion of ODBMS limitations, for a skeptical view of the incorporation of database functionality into existing object-oriented programming languages, and for a defense of an approach that unifies the object-oriented and relational database technologies. Some of these issues were also examined by Stonebraker, in [52] and [53].

## 2.8 Object-Relational Mapping

The systems we have mentioned so far are "pure ODBMSs", in the sense that they employ the object-oriented data model even at the storage level. Systems with similar functionality, but different performance, have also been built on top of relational DBMSs. They are called *object-relational mediators*, or *wrappers*, and perform object-relational mapping: classes are mapped into relational tables, and objects are mapped into tuples. In these systems, objects are stored in tuple-ized form.

An object-relational mediator, along with a relational DBMS, implements an ODBMS. To stress the architectural differences between such ODBMS and a fully object-oriented one, the latter is sometimes referred to as an *OO*DBMS. For a defense of object-relational mapping and a description of Penguin, a research prototype based on this approach, see [24] and [25]. Persistence [26] and DBconnect [35] are examples of commercially available systems in this category.

Rather than using the memory image approach to object storage, mediators perform object/tuple conversion and keep tuple-ized objects in a relational DBMS. At the programming level, however, an object-relational mediator looks like an OODBMS. As in most OODBMSs, a smart pointer scheme is used to integrate persistence into the C++ environment, and objects are cached at the client side, in a way transparent

to the programmer. A tool is provided to generate the object/tuple conversion code, given descriptions of both the object schema and the underlying relational schema.

Ultimately, an object-relational mediator uses SQL to interact with its relational back-end. Because this takes much longer than the client-server interactions in an OODBMS, object caching is yet more crucial for object-relational mediators.

Object-relational mapping is attractive to integrate legacy databases into an object-oriented environment. Some object-relational mediators allow an object structure to be superimposed onto an existing relational database. The superimposed object structure can even include user-defined methods, through which the relational data may be accessed in a fully object-oriented fashion.

# Chapter 3

# ORB/ODBMS Integration

This chapter presents our view of ORB/ODBMS integration. Little has been published about this subject. ODBMSs in the ORB environment are briefly mentioned by the CORBA specification [37], and are discussed to some extent in the appendix B of the ODMG standard [7].

## 3.1 Motivation

The purpose of ORB/ODBMS integration is to allow CORBA objects to be persistently stored in object databases. The resulting system has the strengths of both an ORB and an ODBMS environment:

- The remote method invocation mechanism of the ORB provides application interoperability across different languages, operating systems, and hardware architectures.
- The ODBMS provides object persistence, plus database features such data consistency in the presence of concurrent accesses, crash recovery, and so forth.

ORB/ODBMS integration can be regarded either as a way of supplying CORBA objects with persistence and databases features, or as a way of eliminating or attenuating some ODBMS limitations. The latter perspective leads to a number of reasons for integrating ORBs and ODBMSs:

**Database Heterogeneity.** An ODBMS is a single-vendor solution. Even though most commercially available systems support object databases distributed

across different hardware platforms, all database servers and clients must run ODBMS software provided by the same vendor. ORB/ODBMS integration allows the construction of distributed object databases that are truly heterogeneous, even with respect to the ODBMS software running on the database server nodes.

**Language Heterogeneity.** ORB/ODBMS integration allows access to object databases by CORBA clients written in any language for which a mapping from IDL is defined, thus circumventing the single-language limitation of current ODBMSs.

**"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. Moreover, interfaces can be defined to expose only data items that certain users are permitted to read or update. Hence IDL interfaces to database objects can play a role analogous to relational views[1], both for data independence and for authorization purposes.

**Security.** The remote method invocation mechanism of the ORB requires much less trust in the client than the ODBMS data-shipping approach, which gives database access to the client through its own address space.

If database access is only allowed through IDL interfaces, fine-grained access rights can be implemented using the operation `get_principal`. Encryption-based security can be transparently realized employing either non-standardized features provided by most ORB implementations (*request and reply handlers*), or the interceptors recently standardized in [2].

Implementations of the Security Service [2] specified by the OMG should appear during 1996. ORB/ODBMS integration can make this service available also to database objects.

[1]This analogy only goes to a certain extent. While relational views can be queried like any relational table, the query facilities of the ODBMS are unavailable to IDL clients. The OMG specified an Object Query Service [16] that supports all CORBA objects was specified by the OMG. No implementations of this service exist at the present time. When they become available, "IDL views" may reach the functionality of relational views.

Since it might still be useful to have transient CORBA objects, the integration of an ORB with an ODBMS does not mean that all CORBA objects will be made persistent. Some of the IDL operations requested by a CORBA client may be on persistent objects, while others may be operations on transient objects. Although the CORBA client may need to know whether an object is persistent or not, it is desirable to minimize the effects of object persistence on CORBA clients. They should not need any knowledge of the particular ODBMS used to make an IDL object persistent.

Conversely, not all database objects need to have IDL interfaces and be registered as CORBA objects. For a number of reasons, one might decide to give the status of a CORBA object only to a subset of database objects. *Security* is a possible reason.

Another reason is *data hiding*. If the sole purpose of some database objects is to serve as building blocks in the implementation of higher level ones, only the higher level database objects should have IDL interfaces.

Yet another reason is *performance*. Object database systems are tuned for performance even in the case of fine-grained persistence. They support databases with millions of small primitive objects, and typically allow clustering of related data. If small objects that are usually accessed together are stored together, page-based data shipping is more efficient than operation shipping, because it avoids the cost of interprocess communication at every access to a small object. Thus the CORBA remote method invocation mechanism is unlikely to deliver ODBMS performance on large collections of very small objects. Database performance can be preserved by choosing a suitable subset of database objects, presumably the higher level ones, to be accessed through the ORB.

## 3.2 Process Architecture

Consider a POSIX environment in which object implementations are processes. To store CORBA objects in an ODBMS-managed database, a CORBA server — the process that implements these objects — must have access to the database. In the common case of a client/server ODBMS, this process has to be a database client. Five kinds of processes (see Figure 3.1) may be present in the integrated ORB/ODBMS system:

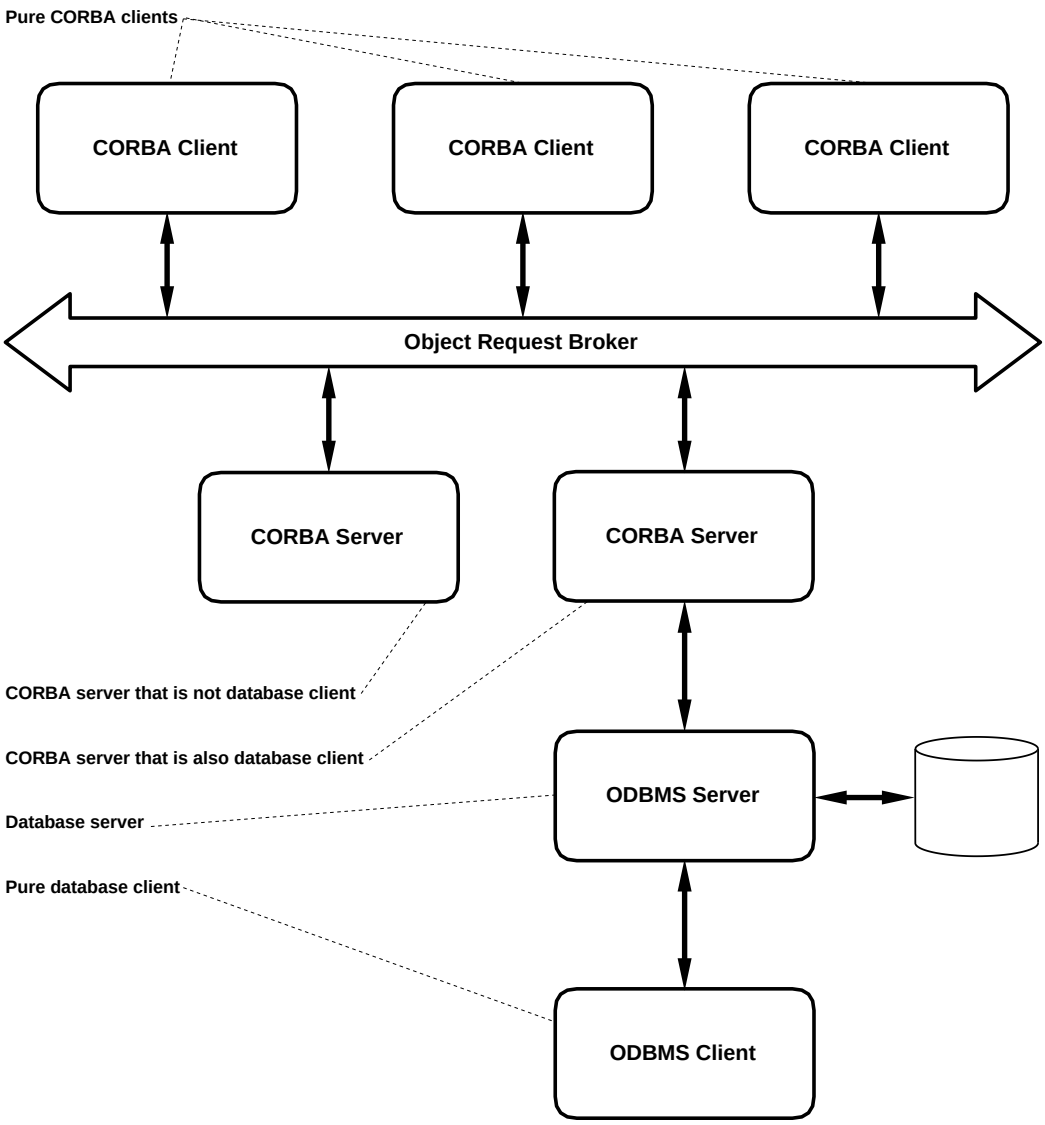



Figure 3.1: Process architecture.

1. Database servers.

   Database server software is typically provided by an ODBMS vendor. ORB-connected multidatabases, comprising object databases from different vendors, can be built by the use of heterogeneous database servers.

2. CORBA servers that are also database clients.

   These are the processes that make direct use of ORB/ODBMS integration to implement persistent CORBA objects. Such a process provides an execution context for the IDL operations on the persistent CORBA objects it realizes. As an ODBMS client, it has access to database objects through its own address space. As a CORBA server, it implements IDL interfaces to some of these objects.

3. CORBA servers that are not database clients.

   These processes may be present to implement transient CORBA objects.

4. Pure CORBA clients.

   User interface processes that do not implement CORBA objects fall in this category. CORBA servers are likely to be also CORBA clients, but we are not considering these as "pure" clients.

5. Pure database clients.

   Database clients that are not CORBA servers perform database access bypassing any IDL interfaces to database objects. Such processes are needed at least for database administration tasks that cannot be accomplished via IDL interfaces. Depending upon a site's security policies (see below), a wider range of trusted user tasks may be performed by pure database clients.

In a networked environment, these processes may be distributed among different nodes. For security reasons, one will typically want to run all database servers and clients on trusted machines. A process on a less trusted machine would only be able to manipulate database objects as a CORBA client, using the IDL interfaces to these objects, subject to an access control mechanism with the desired granularity.

This setting fits the common case of an ODBMS without support for fine-grained authorization.

For performance reasons, one may want to run a CORBA server that is also a database client on the same machine as the corresponding database server. Avoiding the network overhead in the communication between these processes suits the case of very large objects, or the case of a page server managing a database of small objects that are not well clustered according to their usage.

## 3.3 Problems to Solve

Because object persistence is integrated into the C++ programming environment provided by the ODBMS, no read or write commands need to be issued for accessing persistent memory. In spite of this, problems must be solved to make C++ CORBA objects persistent.

### 3.3.1 Why a Direct Approach Cannot Work

At first, it may seem straightforward to make CORBA objects persistent simply by storing their memory representations in an object database. There are three good reasons for not doing so: database space, performance, and object activation.

**Database Space.** Under the common approaches to IDL interface implementation (see Section 1.5), a CORBA object is more than its implementation-defined state. It either inherits all data members, virtual functions, and base classes of the skeleton class that corresponds to the its interface, or is an instance of a tie class. Skeleton and tie classes are derived from IDL-generated interface classes, which typically

1. have virtual member functions that correspond to the interface operations, and
2. use virtual inheritance to mimic the inheritance hierarchy of their IDL interfaces.

Therefore, if the natural mapping of interface inheritance to C++ is used, the following facts are true:

- Every CORBA object inherits data members from the interface class `CORBA::Object`.
- It also holds a pair of "hidden pointers" — a `vbase` pointer and a `vtbl` pointer — for each interface class in its inheritance chain up to `CORBA::Object`.

Depending on the length of this chain and on the space taken by the data members inherited from `CORBA::Object`, the CORBA object can have size noticeably larger than its implementation-defined state. Such overhead in size is especially significant in the case of objects with small state.

**Performance.** ORBs typically use reference counts to implement the operations `duplicate` and `release` (see Section 1.3). Every CORBA object inherits a reference count from `CORBA::Object`. Reference counts are incremented by `duplicate` and decremented by `release`. Having the full memory representation of a CORBA object stored in a database would increase the cost of these operations, because write access to the database would have to be acquired to update the reference count. In the common case of a DBMS with lock-based concurrency control and page lock granularity, this means acquiring an exclusive lock to the database page that contains the reference count.

Moreover, `duplicate` and `release` are automatically called by ORB implementations whenever an operation on an object is invoked by a remote client. Having reference counts stored in the database would result in a substantial decrease in concurrency. Exclusive locks would have to be acquired in all invocations of remote operations, even for operations that do not update an object's state as defined by its implementation. The performance toll would then be twofold, with a high penalty in database concurrency added to the overhead of exclusive lock acquisition at each operation.

**Object Activation.** ORB implementations keep a per-process table of active objects: a new entry is inserted in this table whenever the constructor of a CORBA object is invoked by the corresponding process. In a C++-based ODBMS, however, the constructor of a persistent object is only invoked when the the object is added to the database. Hence, no entries in the table of active objects refer

to CORBA objects constructed and stored by other processes, including previous runs of the same program. After a process exits or crashes, all persistent CORBA objects it created would remain dormant and unreachable through the ORB.

### 3.3.2 Persistence of Implementation Objects

In this section, we consider an alternative to the direct approach. This alternative is not a real solution; we discuss it for better understanding of the problems involved in making CORBA objects persistent, and as a step in the direction of actual solutions. To avoid the database space and performance problems of the direct approach, this scheme does not attempt to make CORBA objects persistent. CORBA objects are always transient, but their implementation objects — instances of implementation classes written by the interface implementor — may be persistent.

Only implementation objects are kept in persistent memory; the CORBA objects associated with these implementation objects are dynamically instantiated in the transient memory of a CORBA server. Before returning a reference to such a CORBA object as a result of a client's request, the server creates the CORBA object. When the client does not need this reference anymore, the server deletes the CORBA object from transient memory. The corresponding implementation object remains in persistent memory.

At different points in time, a persistent implementation object may — and typically will — be associated with different instances of transient CORBA objects. These instances act as "IDL shells" for the implementation object; their purpose is to give CORBA clients access to the IDL methods realized by the implementation object. In such a scheme, a persistent implementation object outlives all CORBA objects it was associated with in the course of its lifetime. This rules out the inheritance approach to interface implementation (see Section 1.5.2), which imposes to an implementation object the same lifetime of its CORBA object. Instead, the relationship between an implementation object kept in persistent memory and a CORBA object is realized through the delegation approach to interface implementation.

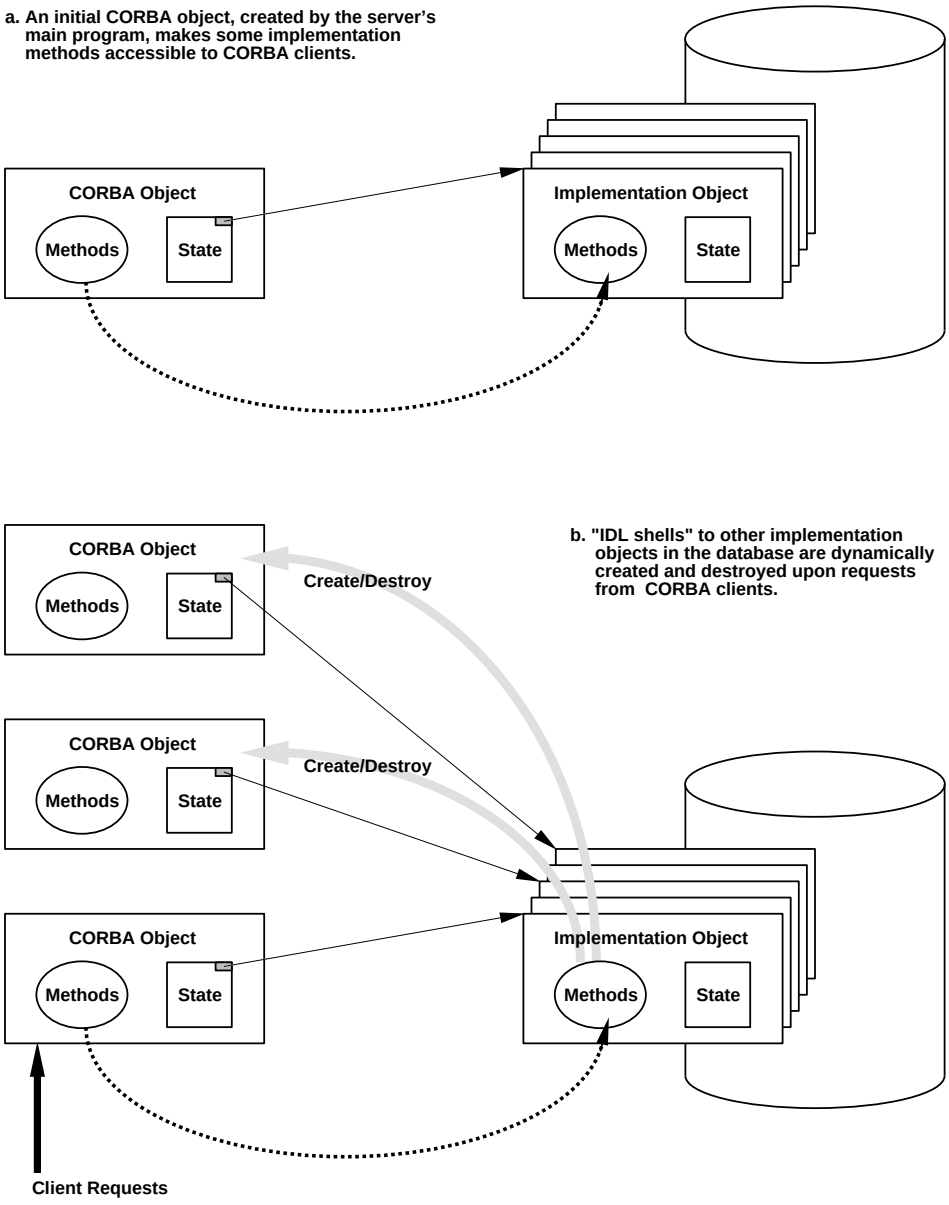



Figure 3.2: Persistence of implementation objects.

### IDL Support to Persistent Implementation Objects

The IDL compiler must support the delegation approach in a way slightly different from the one described in Section 1.5.2, generating tie classes suitable to the association of transient CORBA objects with persistent implementation objects. In the case of a tie from transient to persistent memory, the example of IDL-generated tie class in Figure 1.16 is modified to look like the one in Figure 3.3.

```
// C++
template <class Impl>
class tie_A : public A {
private:
    d_Ref<Impl> ref; // nb:  d_Ref<Impl> and not Impl&
public:
    ...
    tie_A(Impl& impl_obj) :  ref(impl_obj) {}
    CORBA::Short op1() { return ref->op1(); }
    void op2(CORBA::Long l) { ref->op2(l); }
    ...
};
```

Figure 3.3: IDL-generated tie to persistent memory.

Due to the ODBMS restrictions on C++ pointers/references from transient to persistent memory (see Section 2.4), the data member `ref` is not a C++ reference (`Impl&`) anymore; now it is an ODBMS reference (`d_Ref<Impl>`).

### Example: Dynamically Tying a CORBA Object to a Persistent Implementation Object

Consider the IDL operation

```
Patient GetPatient(in Name name) raises(reject);
```

of the `Repository` interface in Figure 1.1. Assume that the CORBA server implementing the `Repository` interface keeps a list of patients in persistent memory. The objects in such list cannot be instances of the interface class `Patient`: since `Patients` are CORBA objects, they are not stored in persistent memory. Instead,

the objects in the patient list are instances of an implementation class for `Patient`, say `Patient_impl`.

When a CORBA client makes a `GetPatient` request, the corresponding method in the server uses the `name` parameter as a key to retrieve `p_i`, a `Patient_impl` instance with that name, from the list of patients. It then executes

```
Patient_ptr p = new tie_Patient<Patient_impl>(p_i);
```

to create a CORBA object that satisfies interface `Patient` and has `p_i` as its implementation object. The return value of the `GetPatient` method is `p`, a CORBA reference to this object. The ORB core transmits the reference `p` back to the client.

This CORBA object must eventually be deactivated. The client is responsible for notifying the server that this object may be deactivated, because it is not needed anymore. For this sole purpose, a new IDL operation is added to the `Repository` interface, say

```
ReleasePatient(in Patient patient);
```

A request to `ReleasePatient` tells the server to deactivate a `Patient` object, but has no effect on its `Patient_impl` counterpart in persistent memory.

### Avoiding Multiple Ties to a Persistent Implementation Object

While a client holds a reference to a CORBA object tied to a persistent implementation object, other clients may obtain CORBA references associated with the same implementation object. A table of active persistent objects can be used to avoid the creation of multiple CORBA objects tied to the same implementation object. This table would contain pairs `(db_ref,orb_ref)`, where `db_ref` is an ODBMS reference to an implementation object that is currently associated with a transient CORBA object, and `orb_ref` is a CORBA reference to this transient object. A new CORBA object associated with a persistent implementation object would only be created if no reference to the implementation object is found in the table. For fast lookup, the table of active database objects would be hashed by `db_ref`s.

**Problems With This Scheme**

The simple scheme we described avoids the drawbacks of the direct approach, but sacrifices most benefits that a real integration between ORB and ODBMS should realize. The problems with mere persistence of implementation objects are:

1. CORBA references that correspond to persistent implementation objects are not persistent themselves; they are only valid for short periods of time. Clients are prevented from storing CORBA references in persistent media. This disallows the construction of ORB-connected multidatabases.

2. The inclusion of operations such as `ReleasePatient` changes the IDL interfaces seen by clients. A client must be aware that a particular CORBA object has a persistent implementation object.

3. An operation such as `ReleasePatient` is actually a memory management operation that deallocates the memory space a CORBA object takes in its server. These operations transfer to the clients the responsibility of managing the server's memory; they give "buggy" clients the ability to cause server memory leaks.

Even for someone willing to live with problems 1 and 2, problem 3 alone subtracts all viability from our naive scheme for persistence of implementation objects: remote management of server memory is frontally antagonistic to the idea of stateless servers, a concept widely accepted due to its positive effects on the robustness of any distributed system. Despite its problems, persistence of implementation objects is a step in the direction of practical approaches to persistence of CORBA objects.

## 3.4 The Object Database Adapter

Rather than tackling at the application level the issues identified in the previous section, we follow the ODMG proposal [7] and introduce an object adapter — the Object Database Adapter – that deals with those issues in an application independent way.

The object adapter is a replaceable component of the CORBA architecture. Only the Basic Object Adapter is currently specified by CORBA, but "it was envisaged

that over time additional, less general-purpose, adapters would be standardized by the OMG, such as for use with objects stored in OODBs (an Object Database Adapter), or for in-process objects whose implementations are maintained in dynamically linked libraries (the Library Object Adapter)" [30].

Note, however, that only the ORB implementor can actually provide a replacement for the BOA: the object adapter interacts with the ORB core and with the skeletons through ORB-dependent interfaces (Figure 1.2). Information on these non-standard interfaces is unavailable to users.

Fortunately this is not a real problem. Instead of replacing the BOA, an Object Database Adapter can built as an add-on to the BOA, as a library that uses and extends BOA services. In this architecture, depicted in Figure 3.4, the ODA works together with the BOA on the generation and interpretation of references to persistent CORBA objects. Since the ODA consists of a library linked to the object implementation, it interacts with the BOA through the ORB-independent interface the BOA makes available to object implementations.

The ODA solves the problems identified in Section 3.3.1 by using the delegation (tie) approach to interface implementation. Only implementation objects are stored in the database. The corresponding tie objects are automatically instantiated by the ODA whenever they are needed and released when not needed.

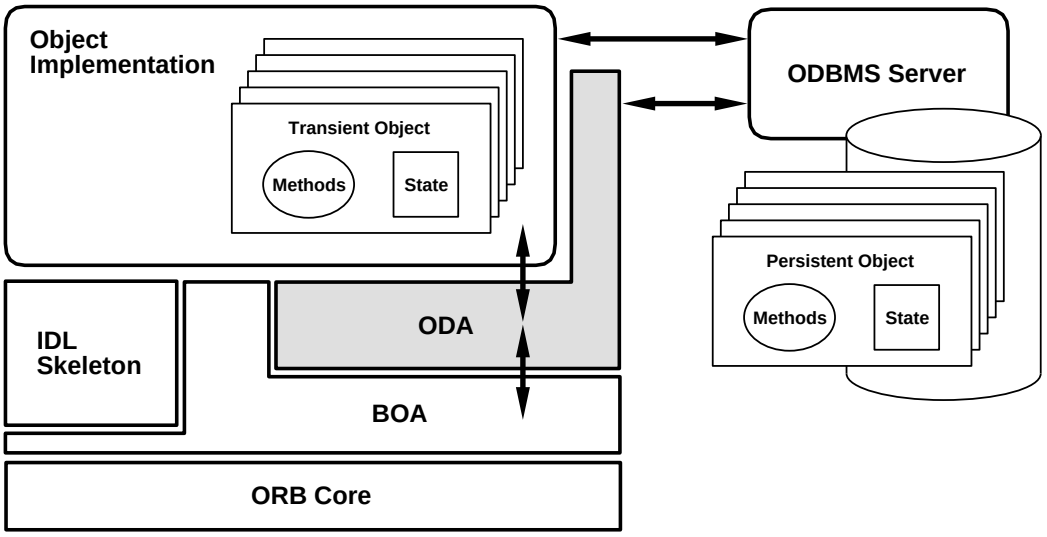



Figure 3.4: The Object Database Adapter.

This approach looks similar to our naive scheme for persistence of implementation objects. In that scheme, however, the object implementation was responsible for instantiating and deleting tie objects. By assuming these responsibilities, the ODA

- provides persistence to CORBA references (tie objects are automatically instantiated when need), and
- removes the need for remote management of server memory (tie objects are automatically released).

To allow automatic instantiation of tie objects, the ODA embeds a stringfied ODBMS reference to the corresponding implementation object into the `id` (also known as `ReferenceData`) field of a CORBA reference to a persistent object. Moreover, it provides an object activation routine that builds the tie object given an ODBMS reference to its implementation object.

Every access to a persistent object must be performed within a transaction. In the absence of an external monitor providing the Object Transaction Service [39], the ODA enforces this rule by automatically starting a transaction before each IDL-defined operation on a persistent object begins execution (if there is no transaction already active) and committing (or aborting) the auto-started transaction at the end of the operation.

Finally, it is also desirable that the ODA makes CORBA references transparently storable. In the following sections, we discuss two approaches to the construction of an ODA. The first one, which we call *pseudopersistence*, makes object references persistent, but not transparently storable. The second approach, which we call *full persistence*, provides transparent storability to object references, both in the case of references to local objects and in the case of references to remote objects. Full persistence comes in two flavors: *smart pointer-based persistence* and *virtual persistence*; the latter applies only to the case of a virtual memory-based ODBMS.

In what follows, the environment provided by the ODA to an object implementation will be depicted as in Figure 3.5, just to avoid heavily populated figures. The reader should keep in mind that this is a simplified representation of the architecture in Figure 3.4: besides involving an ODA, "persistence in a CORBA server" also involves an ODBMS server.

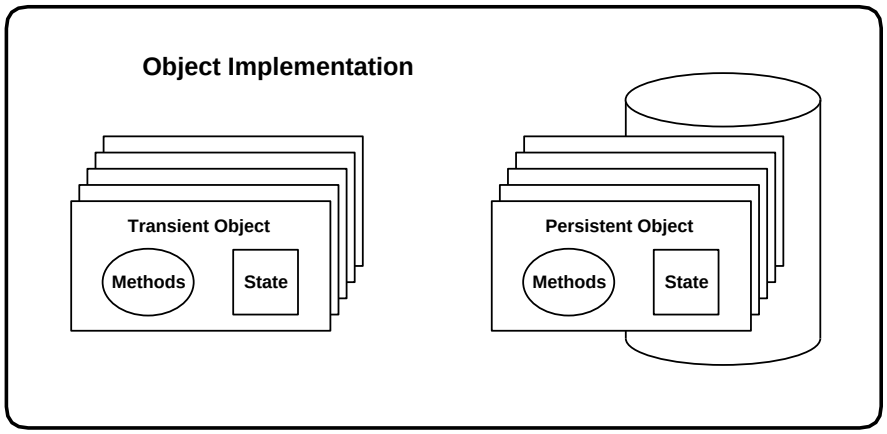



Figure 3.5: Simplified representation: "persistence in a CORBA server."

## 3.5 Pseudopersistence

What we call *pseudopersistence* is a combination of two ideas: persistence of implementation objects and persistence of CORBA object references. CORBA references are made persistent through the basic scheme outlined on page 25. As we discussed there, that scheme left some important issues unresolved. To address those issues, we assume an ORB such as Orbix, with a well defined object activation mechanism (see 1.6), plus request and reply handlers (see 1.7).

Figure 3.6 illustrates the pseudopersistence approach, which works as follows:

1. The delegation approach is used to dynamically associate transient CORBA objects with persistent implementation objects. These ORB objects are ties from transient memory to persistent memory; we call them *pseudopersistent objects*.

2. The ODA implements *reference embedding*. Whenever a CORBA reference to a pseudopersistent object is created, the ODA assigns it a special `id` value. A field in the `id` holds the information that the object is pseudopersistent. Other field specifies the identity of the corresponding implementation object, in the form of an ODBMS reference converted to a string. CORBA references to pseudopersistent objects can only be created by the ODA, in order to ensure that any such reference has an ODBMS reference to the appropriate implementation object embedded in its `id`.

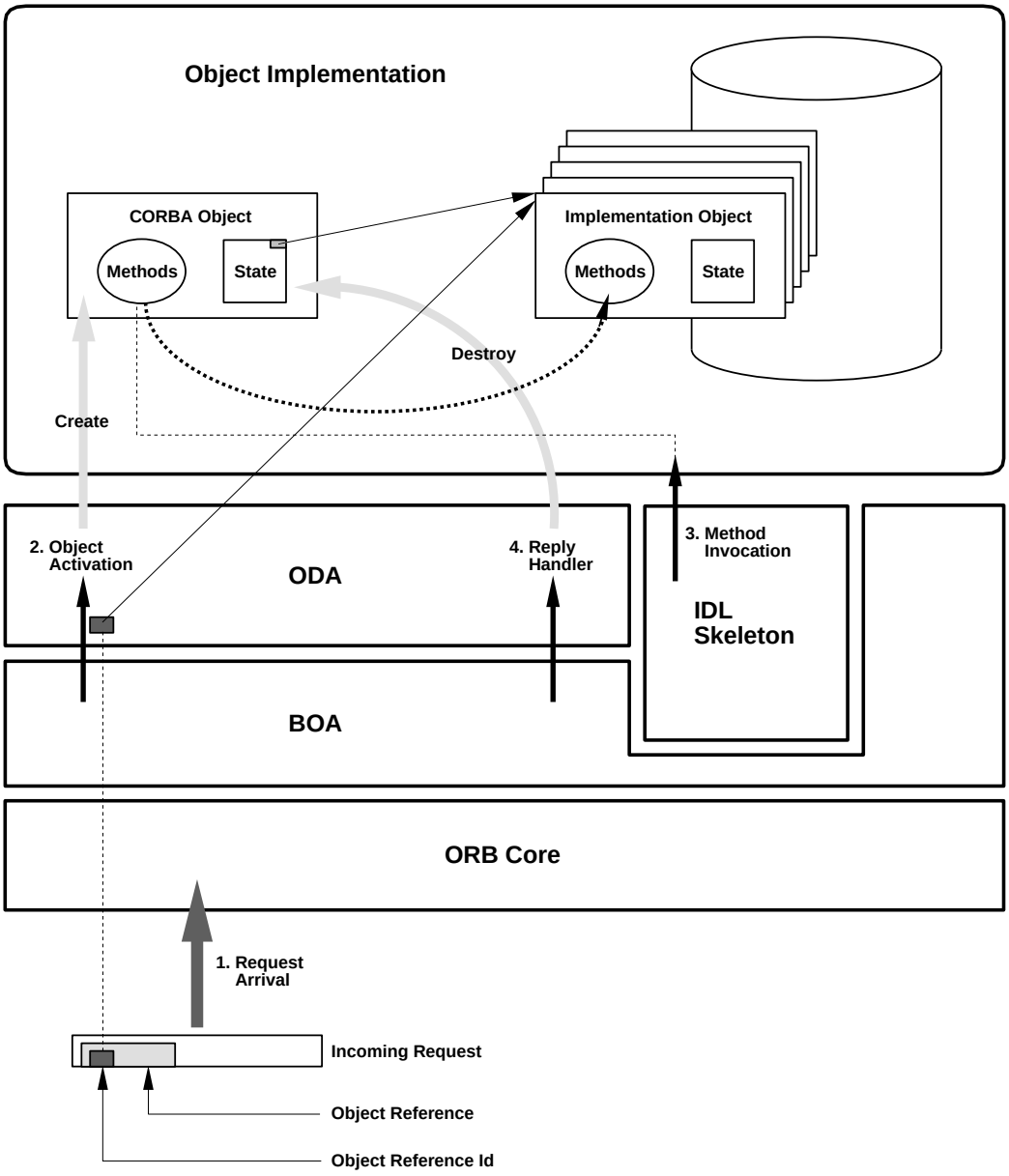



Figure 3.6: Incoming request handling in the pseudopersistence approach.

3. The ODA provides a special object activation routine. When a request to a pseudopersistent object that is not active[2] is received, this routine instantiates the object. The incoming request contains a marshaled reference[3] to its target; the ORB passes to the object activation routine the `id` value of the marshaled reference. The object activation routine uses this parameter to instantiate the pseudopersistent object as a tie to the implementation object specified by the ODBMS reference embedded in the `id`. It also constructs an actual CORBA reference to the pseudopersistent object, and assigns to this reference the `id` of the marshaled one. This CORBA reference, converted to a reference to `CORBA::Object`, is the return value of the object activation routine.

4. The object activation routine is also called when an object reference enters the server's address space not as the target of a request, but in any of the remaining ways listed in Section 1.6. In each case, it performs actions similar to the ones just described in 3.

5. The ODA provides a special reply handler. This handler deactivates (deletes) all pseudopersistent objects whose instantiations were triggered by the current request. To allow this, the ODA maintains a "to release" list, a list of references to pseudopersistent objects to be released at the end of the current operation. In addition to what was described in 3, the object activation routine inserts into the "to release" list references to each pseudopersistent object instantiated during a request service.

6. Whenever an implementation class creates a pseudopersistent object to be passed to a client, either as the return value or as an output parameter of a request, it does so by calling a ODA function that inserts a reference to the pseudopersistent object into the "to release" list.

The net result of 5 and 6 is that pseudopersistent objects passed back and forth between a server and its clients, either as parameters or as return values of requests, have short existence: they only exist while a request is being serviced. At the end

[2] "Not active" means "does not exist in the server address space at the present time".

[3] At this time, this CORBA reference only exists in marshaled form.

of every request service, the reply handler removes these objects from the server's address space. If such a pseudopersistent object is needed again, in a future request service, an equivalent to it will be instantiated by the object activation routine.

Note that the reply handler provided by the ODA has to be a post-marshal handler: because the return value and the output parameters of a request must be available to the marshaling routine, they can only be released after the reply is marshaled.

To avoid multiple ties to an implementation object (see discussion in page 63) and allow efficient implementation of a function that returns the CORBA reference to a pseudopersistent object given its implementation object, the ODA maintains an in-memory table of active pseudopersistent objects. The entries of this table are pairs `(db_ref,orb_ref)`, where `orb_ref` is a CORBA reference to a currently active pseudopersistent object, and `db_ref` is an ODBMS reference to the corresponding implementation object. For fast lookup, the table of active pseudopersistent objects is hashed by `db_ref`s.

Pseudopersistent objects cannot be directly instantiated by a server. The server obtains CORBA references to pseudopersistent objects via calls to the ODA. In such a call, an input parameter specifies an implementation object. If this implementation object appears in an entry of the table of active pseudopersistent objects, the ODA returns the associated CORBA reference. Otherwise the ODA:

- creates a new tie for the implementation object,
- embeds a stringfied ODBMS reference to the implementation object into the `id` of the newly created CORBA object,
- inserts a new pair `(db_ref,orb_ref)` into the table of active pseudopersistent objects,
- inserts a reference to the newly created CORBA object into the "to release" list, and
- returns this CORBA reference.

The destructor of a pseudopersistent object removes the corresponding entry from the table of active pseudopersistent objects. Its constructor is private, to ensure that only the ODA can instantiate persistent objects.

### Caching Pseudopersistent Objects

There is no reason to release *all* active pseudopersistent objects at the end of every operation; the ODA only needs to ensure that these objects will be eventually released. Successive reinstantiations of pseudopersistent objects can be avoided with the following caching scheme:

- The "to release" list is implemented by a FIFO queue. At any point in time, the number of active pseudopersistent objects (`n_active_ppobjs`) is the cardinality of this list.
- Rather than removing every object from the "to release" list and releasing them all, the reply handler only does this only for the first `n_active_ppobjs - cache_size` objects in that list.

Does a such simple caching scheme yield a noticeable performance gain? We will answer this question in Chapter 5.

### Comparison with Simple Persistence of Implementation Objects

Pseudopersistence removes most problems presented by mere persistence of implementation objects:

- Clients do not have to manage the server's memory. Pseudopersistent objects are instantiated and released automatically by the Object Database Adapter.
- The pseudopersistence of an object has no effect on its IDL interface.
- CORBA references to pseudopersistent objects are made persistent by the object activation mechanism. Clients can freely store these references in persistent media; this allows the construction of ORB-connected multidatabases.

Pseudopersistence should also perform better than simple persistence of implementation objects. In that scheme, clients control the lifetimes of transient ties to persistent implementation objects, and thus are able to avoid reinstantiations of these ties. Even so, the tie instantiations avoided by a client written for performance are outweighed by the cost of the remote method invocations with the sole purpose of managing

the server's memory. Pseudopersistence does not incur the cost of remote memory management.

One limitation, however, remains: CORBA references to pseudopersistent objects are not transparently storable. They must be converted to strings to be stored.

### The Limitation of Pseudopersistence

Because CORBA references to pseudopersistent objects are made persistent through the object activation mechanism, their persistence is not effective in every situation, but only in those covered by this mechanism.

Object activation is performed when an CORBA reference enters an address space, in any of the ways listed in Section 1.6. The retrieval of a CORBA reference from an object database does not appear in that list: a reference just retrieved from an ODBMS-managed database is not considered to be entering the address space of the database client. This is in agreement with the view of an ODBMS extending the address space of the database client to include the objects kept in persistent memory. In this perspective, anything retrieved from an object database does not enter the address space of the ODBMS client — it was already in the client address space even before being retrieved.

Object relationships can be represented through object references converted to strings. Conceptually, a reference stored in string form does not exist as a CORBA object reference; it only enters the address space of the database client when this process calls the ORB operation `string_to_object`.

Although possible, the explicit representation of object relationships by CORBA references converted to strings is a clumsy way to organize an object database: having to call `string_to_object` to access a related object contradicts the ODBMS efforts to smoothly integrate persistence into the programming language environment. In the absence of transparent storability of CORBA references, the relationships between pseudopersistent objects *within a server* are best represented by ODBMS references between the corresponding implementation objects. Stringfied CORBA references must be used for *cross-server relationships*.

## 3.6 Full Persistence

What we call *full persistence* extends pseudopersistence by providing a transparent mechanism for the storage and retrieval of CORBA references. This mechanism makes these references look like any other database item to the programmer, hiding any format conversions that may take place when a CORBA reference is stored, when it is retrieved, or when it is used.

This extension makes a pseudopersistent object appear as one truly stored in the database. Even though the object is not actually kept in persistent memory, we will call it persistent.

A *persistent CORBA object* is a transient tie to a persistent implementation object, but it can be referenced like an object actually stored in an ODBMS-managed database. References to persistent CORBA objects are generated and interpreted by the ODA, which provides them with two distinct attributes:

**Persistence in the CORBA sense.** CORBA clients can store references to persistent CORBA objects, and may use them at any time without warning. The pseudopersistence approach accomplishes this.

**Transparent storability.** Object relationships can be represented by transparently stored references to persistent CORBA objects. Transparent storability, or "persistence in the ODBMS sense", is what full persistence adds to pseudopersistence.

### 3.6.1 Smart Pointer-Based Persistence

In the case of an ODMG-compliant ODBMS, full persistence can be achieved through an extension the pseudopersistence approach. To realize full persistence, the ODA provides pseudopersistence, and augments it with a smart pointer scheme that makes CORBA references transparently storable.

The pseudopersistence strategy described in pages 67–69 remains in effect, with the substitution of "pseudopersistent object" by "persistent CORBA object" in items 1–6 of that description. The ODA complements that strategy with the smart pointer scheme outlined in what follows.

### Storing CORBA References as Smart Pointers

In a smart pointer-based ODBMS, relationships between database objects are represented by ODBMS references, which in turn are implemented through a smart pointer scheme. A similar smart pointer scheme is used by the ODA to store CORBA references in an object database. The basic idea is to store a reference in way such that the extra actions taken when the reference is followed are made invisible to the programmer: these actions happen as a side effect of dereference.

A reference to a persistent CORBA object whose interface class is `I` is stored as an instance of an ODA-defined template class `ODA_Ref<I>`. If the referenced object is remote (lives in another process), a data member of `ODA_Ref<I>` holds a stringfied CORBA reference to the object.

Stringfied CORBA references could also be used to store references to persistent CORBA objects that are local. In this case, however, better performance and less overhead in database space are achieved with the use of another data member of `ODA_Ref<I>`, this one holding an ODBMS reference to the appropriate implementation object within a database accessed by the CORBA server.

Note that these CORBA references cannot be stored in their native form, because the persistent CORBA objects they reference are actually transient ties: their lifetime is restricted to the current request service, as in the pseudopersistence approach.

Overloading of the C++ operators "`->`" and unary "`*`" by `ODA_Ref<I>` ensures that CORBA references stored in these forms will have effectively entered the process address space by the time they are dereferenced. For a reference to a remote object, the dereference operators convert the string present in the `ODA_Ref<I>` instance back to a CORBA reference, via `string_to_object`. For a reference to a local object, these operators call the ODA to obtain a tie to the implementation object identified by the ODBMS reference in the `ODA_Ref<I>` instance.

The ODA inserts into the "to release" list references to each CORBA object that entered the process address space via side effects of the C++ dereference operators. As in the pseudopersistence approach, a reply handler provided by the ODA releases these objects at the end of a request service. The caching scheme discussed in page 71 is applicable here as well.

### Implementing ODA References

Figure 3.7 shows a possible implementation of the template class `ODA_Ref<I>`. Its dereference operator looks like the one presented in Figure 3.8. Since we want to store ODA references in databases, we made the class `ODA_Ref<I>` persistent capable, by deriving it from `d_Object`. The constructor of an ODA reference initializes either

```
// C++
template <class I>
class ODA_Ref :  public d_Object {
private:
    I_ptr p;             // CORBA reference to an object of interface class I
    char* orb_ref;       // stringfied CORBA reference to this object
    d_Ref_Any impl_ref;  // ODBMS reference to the impl.  object
public:
    oda_Ref(I_ptr corba_ref);
    virtual ~oda_Ref();
    I_ptr operator->() const;
    I& operator*() const;
    I_ptr corba_ref() const;
    ...
};
```

Figure 3.7: The `ODA_Ref<I>` template class.

the data member `orb_ref` (if the referenced object is remote) or the data member `impl_ref` (if the referenced object is local) to a non null value.

If no further precautions are taken, a problem arises when instances of class `ODA_Ref<I>` are stored in a database: the dereference operation performs a test for nullity `CORBA::is_nil(p)`. Even though `p` is intended to point to a transient CORBA object, this test is using the *stored value* of `p`. The last value assigned to `p` (possibly by another process) is the one being tested for nullity.

Fortunately the ODMG specification provides a way to solve this problems. The relevant member functions of `d_Object`, the persistent capable root class, are shown in Figure 3.9.

An application calls `mark_modified` to request an update of the object's stored image. If this call is not made, changes to the object are not guaranteed to be

propagated to the database. (This may or may not happen, and may depend on other `mark_modified` calls, e.g.: a `mark_modified` call on some object in the same database page.)

The `d_activate` and `d_deactivate` functions are called *by the ODBMS*, not by the application. Their purpose is to support transient pointers (pointers to transient memory) as data members of a persistent object. Transient pointers must be managed as the persistent object enters and exits the application cache: memory may need to be allocated and deallocated when these events occur, for instance. A `d_activate` call is performed whenever the object enters the application cache, and a `d_deactivate` call whenever the object leaves the application cache. Figure 3.10 illustrates the calls made in the course of an object's lifetime.

Note that the constructor of a persistent object is only called when the object is first added to the database, and the destructor when the object is deleted from the database. Between the object's construction and its destruction, as it leaves and reenters the application cache, a series of `d_deactivate` and `d_activate` calls is performed by the ODBMS.

It is now clear how to solve the ODA reference problem: by redefining the virtual function `d_activate` in class `ODA_Ref<I>`, we ensure that the data member `p` of an ODA reference is set to null when the ODA reference enters the application cache. Since changes in the value of `p` do not need to be propagated to the database, dereference operations perform assignments to this data member, but do not

```
// C++
template <class I>
I_ptr ODA_Ref<I>::operator->() {
    if (CORBA::is_nil(p)) {
        ...     // uses orb_ref or impl_ref (whichever is non nil)
                // to obtain objref, a CORBA reference to the object
        p = objref;
    }
    return p;
}
```

Figure 3.8: Class `ODA_Ref<I>`'s dereference operator.

call `mark_modified`.

### 3.6.2 Virtual Persistence

The strategy employed to augment pseudopersistence in the case of an ODMG-compliant ODBMS may not work for a virtual memory-based one. In a virtual memory-based ODBMS, the memory representation of an ODBMS reference is a plain C++ pointer, the virtual memory address that results from the swizzling of an OID at page fault time. The ODBMS may expect interobject relationships to be represented in memory by plain C++ pointers. Some of its components — the ones the support collections and queries, for example — may assume that interobject relationships are expressed in this way.

If this is the case, CORBA references must therefore be represented as plain C++ pointers, and not as instances of a special class defined by the ODA. This rules out a smart pointer-based strategy, because C++ supports redefinition of operators only on classes, and not on native data types, such as pointers.

If a CORBA reference is stored as a plain C++ pointer, this pointer must point to some persistent object. The problem is that the "persistent" CORBA object identified by the reference is actually a transient tie. The ODA solves this problem by introducing a persistent *representative* of this object. References to persistent CORBA objects are stored as C++ pointers to the persistent representatives of these

```
// C++
class d_Object {
public:
    d_Object();
    virtual ~d_Object();
    ...
    void mark_modified(); // mark the object as modified
    virtual void d_activate(); // called on entry into the application cache
    virtual void d_deactivate(); // called on exit from application cache
    ...
};
```

Figure 3.9: The `d_Object` class.

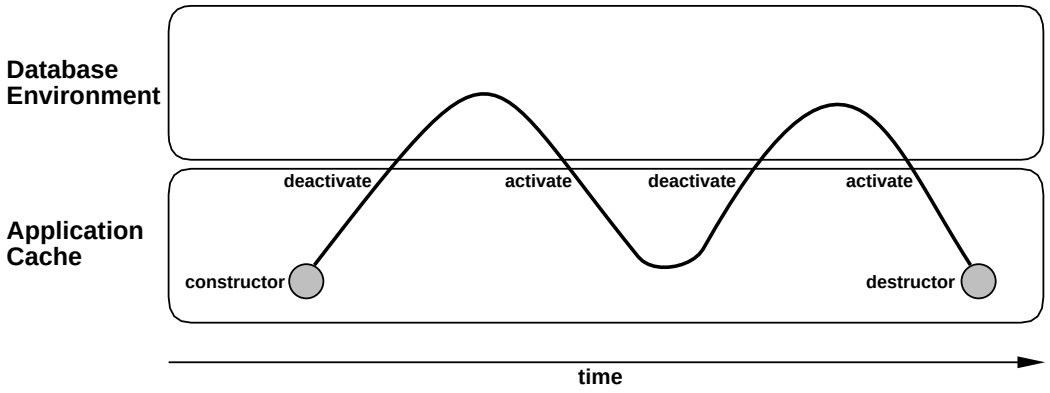



Figure 3.10: Persistent object activation and deactivation.

objects. Since representatives are actually stored in the database, their size should be as small as possible. A representative should contain only the minimum amount of information required for the identification and for the instantiation of the persistent CORBA object it represents.

The actual CORBA objects identified by the CORBA references present in the database and represented by the persistent representatives these references point to must be instantiated at some point in time. Our goal of transparency excludes the possibility of placing this burden on the programmer: the ODA is supposed to instantiate these CORBA objects automatically. The solution is to perform such instantiation at page fault time.

At page fault time, just before it gives the database client access to a faulting page allocated to persistent memory, the ODBMS goes through the objects in the page. It swizzles OIDs to virtual memory pointers, and also performs other tasks, such as the initialization of any `vtbl` pointers in the page. This would be a convenient time to process representatives of persistent CORBA objects. The ODA would then instantiate an actual CORBA object for each representative in the faulting page, and would also provide transparency by somehow "swizzling" these representatives to actual CORBA objects.

We call such an approach *virtual persistence*. To support it, the ODBMS must give database clients the possibility of specifying functions to be executed at page fault time.

### Page Fault Hooks

The interfaces described here are present in ObjectStore, the only commercially available ODBMS that is virtual memory-based. Database clients can install *access hooks*, functions to be executed either when objects of specified classes in specified databases become accessible to the client, or when these objects become inaccessible to the client. The client installs an access hook at runtime, specifying a particular database and a particular class.

The access hooks executed when objects become accessible are called *inbound hooks*. For each transaction, a given inbound hook is called the first time an instance of the specified class in the specified database is accessed by the client, as well as each time such instance is transferred into the client cache. These calls take place at page fault time: when a page of the specified database faults, the inbound hook is called for each instance of the specified class in the faulting page.

The access hooks executed when objects become inaccessible are called *outbound hooks*. A given outbound hook is called whenever an instance of the specified class in the specified database is found in the client cache at the end of a transaction, as well as each time such instance is transferred out of the client cache within a transaction.

An access hook is passed the address of the page that becomes accessible or inaccessible. It can write to this page without causing a disk write operation, by calling a special "hidden write" function that updates the page in the client cache without write-locking it, and without marking it as modified.

### Representatives of Persistent CORBA Objects

We now present a scheme to represent persistent CORBA objects in a database managed by a virtual memory-based ODBMS. This scheme introduces minimum overhead in database space.

Consider again the IDL interface A, shown in Figure 1.9, and its IDL-generated interface class, shown in Figure 1.10. The corresponding tie template, generated by an IDL compiler that supports virtual persistence, looks like the one shown in Figure 3.11. Note that the inheritance from the interface class `A` is now virtual, and that all functions corresponding to IDL operations are virtual member functions; these facts are crucial and will be used shortly. Also note that the data member `ref` is not

```
// C++
template <class Impl>
class tie_A : public virtual A {
private:
    Impl& ref;
public:
    ...  // ORB-dependent member functions go here
    tie_A(Impl& impl_obj) :  ref(impl_obj) {}
    virtual CORBA::Short op1() { return ref.op1(); }
    virtual void op2(CORBA::Long l) { ref.op2(l); }
    ...
    virtual ~tie_A();
};
```

Figure 3.11: IDL-generated tie for virtual persistence.

an ODBMS reference (`d_Ref<Impl>`) anymore, but a plain C++ reference (`Impl&`): a virtual memory-based ODBMS allows C++ pointers/references from transient to persistent memory to be used within a transaction.

Figure 3.12 shows how a typical C++ compiler lays out a `tie_A<A_impl>` in memory. Memory layout of C++ objects is compiler dependent; virtual persistence only relies on the inherited subobjects (or the `vbase` pointers to the virtually inherited ones), the `vtbl` pointer, and the data members of a C++ object being all laid out contiguously, in a fixed order (which needs not be the one shown in Figure 3.12), at the beginning of the memory area allocated to the object. Most if not all C++ compilers lay out objects in this way.

The `A_part` of the object is what takes the most space. It contains all data members inherited from `CORBA::Object`, plus `vbase` and `vtbl` pointers for each interface class in its inheritance chain up to `CORBA::Object`. In this particular example, the interface class `A` is directly derived from `CORBA::Object`. In the general case, however, several interface classes may appear in such inheritance chain.

The only information required to construct a `tie_A<A_impl>` instance is the address of the corresponding implementation object. A representative of a `tie_A<A_impl>` must contain this information. The `A_part` of the object needs not be in the representative. To allow swizzling of the representative to an actual

```
// Memory layout of an object of class tie_A<A_impl>,
// represented as a C++ structure

struct {

    // inherited subobjects
    //  (There is only one, of class A. Due to virtual
    //  inheritance, just a pointer to it goes here.)

    void* vbase;    // points to the A part of the object

    // virtual table pointer

    void* vtbl;     // points to the virtual table of tie_A<A_impl>

    // data members

    Impl& ref;      // this is the only data member

    // virtually inherited subobjects
    // (They may go here, or may be allocated separately;
    // it does not matter.)

A A_part;       // the A part of the object may be elsewhere

};
```

Figure 3.12: Memory layout of a `tie_A<A_impl>` instance.

`tie_A<A_impl>` instance, we set aside slots for `vbase` and `vtbl` pointers within the representative, in the same relative positions these pointers take within the memory layout of a `tie_A<A_impl>` instance.

Figure 3.13 shows the class definition for the representative of a persistent CORBA object that satisfies the interface A and is local to the server holding a persistent reference to the object. The `ref` field is used to express the database relationship between the representative of a persistent CORBA object and the corresponding implementation object. Unlike `ref`, the fields `vbase` and `vtbl` are never really written to the database. Even though they take database space, their purpose is not one of

```
template <class Impl>
class tie_A_representative {
public:
    void* vbase;    // points to the A part of the object
    void* vtbl;     // points to the virtual table of tie_A<A_impl>
    Impl* ref;      // points to the implementation object
};
```

Figure 3.13: Representative of an object of class `tie_A<Impl>`.

keeping the addresses of the virtual base and of the virtual table in persistent memory, but one of allowing the representative to be swizzled to the actual object it represents.

### Swizzling Representatives at Page Fault Time

The ODA provides an inbound and an outbound hook for each representative class. At page fault time, the inbound hooks instantiate the persistent CORBA objects whose representatives live in the faulting page, using the the `ref` fields of the representatives. The inbound hooks also *swizzle* the representatives to the actual objects.

Swizzling a representative means copying the `vbase` and `vtbl` pointers of the actual object into the corresponding fields of the representative. The inbound hooks perform such copies using the "hidden write" function, which neither causes a disk write operation nor requires a write-lock to the database page.

Once swizzled, representatives can be used exactly like the objects they represent. No further runtime conversions or checks are required:

- Since all member functions of a persistent CORBA object are virtual, any calls to these functions use the `vtbl` pointer of the object. This pointer is present in the swizzled representative, with the same value, and in the same relative position.

- Since a persistent CORBA object virtually inherits a subobject of the IDL-generated interface class corresponding to its interface, any data members directly or indirectly inherited from the interface class are accessed only through the `vbase` pointer of the object. This pointer is also present in the swizzled

representative, with the same value, and in the same relative position.

- The single non-inherited data member of a persistent CORBA object is `ref`. This data member also present in the swizzled representative, with the same value, and in the same relative position.

When the representatives become inaccessible, the outbound hooks release the current instances of the represented objects.

As a concluding remark on this subject, notice that we have assigned an additional meaning to the word "swizzling". In the ODBMS field, this word refers to *pointer swizzling*, the translation of OIDs to virtual memory addresses, typically performed in-place. Pointer swizzling is done by the virtual memory-based ODBMS anyway, and is implicit in the scheme we have just described: recall that representatives were introduced to allow the storage of references to (non-stored) persistent CORBA objects as C++ pointers, which point to the actually stored representatives of these objects. The ODA needs not be concerned with the swizzling of these pointers; the ODBMS takes care of it. What we introduced is an additional in-place translation, also performed at page fault time, but of a different nature: the conversion of representatives to the actual objects they represent. This one is not performed by the ODBMS, but by the ODA. Rather than a swizzling of *pointers*, such conversion is in fact a swizzling of *objects*.

### Activation of Persistent CORBA Objects: Internal and External

A persistent CORBA object may be activated by an internal action taken by the server — a virtual memory access to the object's representative — or by an incoming request containing a CORBA reference to the object. Figure 3.14 illustrates the first case. When executing an operation requested by a client, the server performs a memory access to an object representative (possibly by following a reference between persistent objects in the database), which causes the instantiation of a persistent CORBA object. At the end of the operation, an outbound hook deletes the object.

Figures 3.15, 3.16, and 3.17 show the sequence of events that takes place when an incoming request activates a persistent CORBA object. The incoming request contains a CORBA reference to its target, a persistent object. The reference `id`, which contains a stringfied ODBMS reference to the corresponding implementation object,

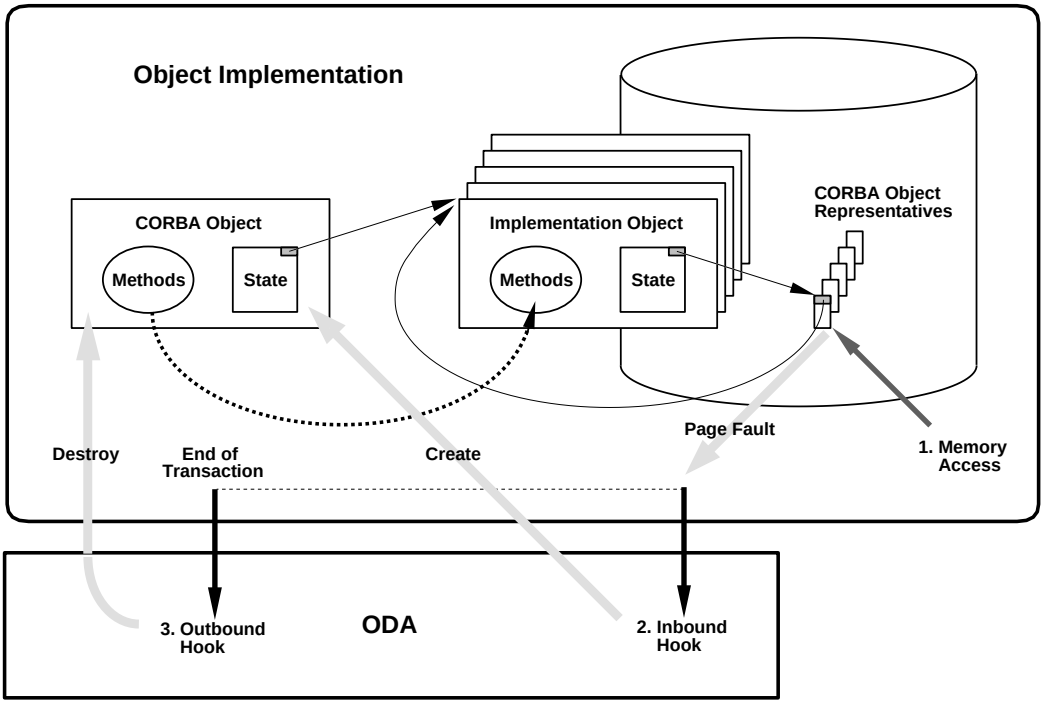



Figure 3.14: Virtual persistence.

is passed to the object activation routine (Figure 3.15). This routine obtains the address of the target representative and performs a memory access to it, causing the instantiation of the target object (Figure 3.16). The request is then delivered to the newly instantiated object, through an up-call from the skeleton. (Figure 3.17). While servicing the request, the server may access other object representatives and cause further instantiations of persistent CORBA objects. At the end of the operation, an outbound hook deletes all these instances, including the target object.

### Persistent References to Remote CORBA Objects

Our scheme for transparently storing references to persistent CORBA objects works only if these objects are *local* to the CORBA server. For references to *remote* CORBA objects, some changes to that scheme are necessary.

Consider again the IDL interface A, shown in Figure 1.9, and its IDL-generated interface class, shown in Figure 1.10. A C++ client accesses this interface through a *stub object*, an object of a *stub class* generated by the IDL compiler. Figure 3.18

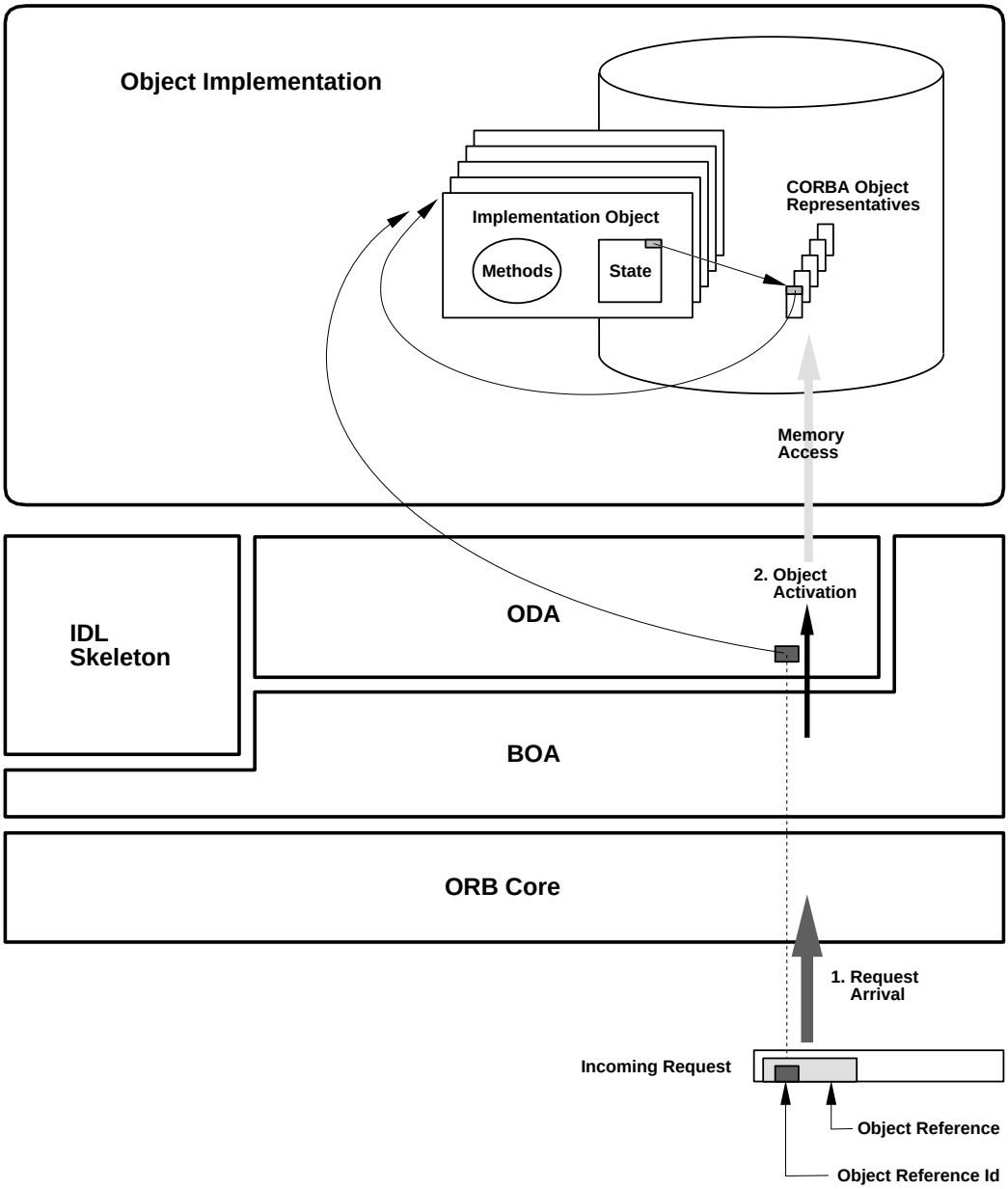



Figure 3.15: Incoming request handling in virtual persistence (part I). A request to a persistent CORBA object arrives at the server. The object activation routine performs a memory access to the representative of the target object.

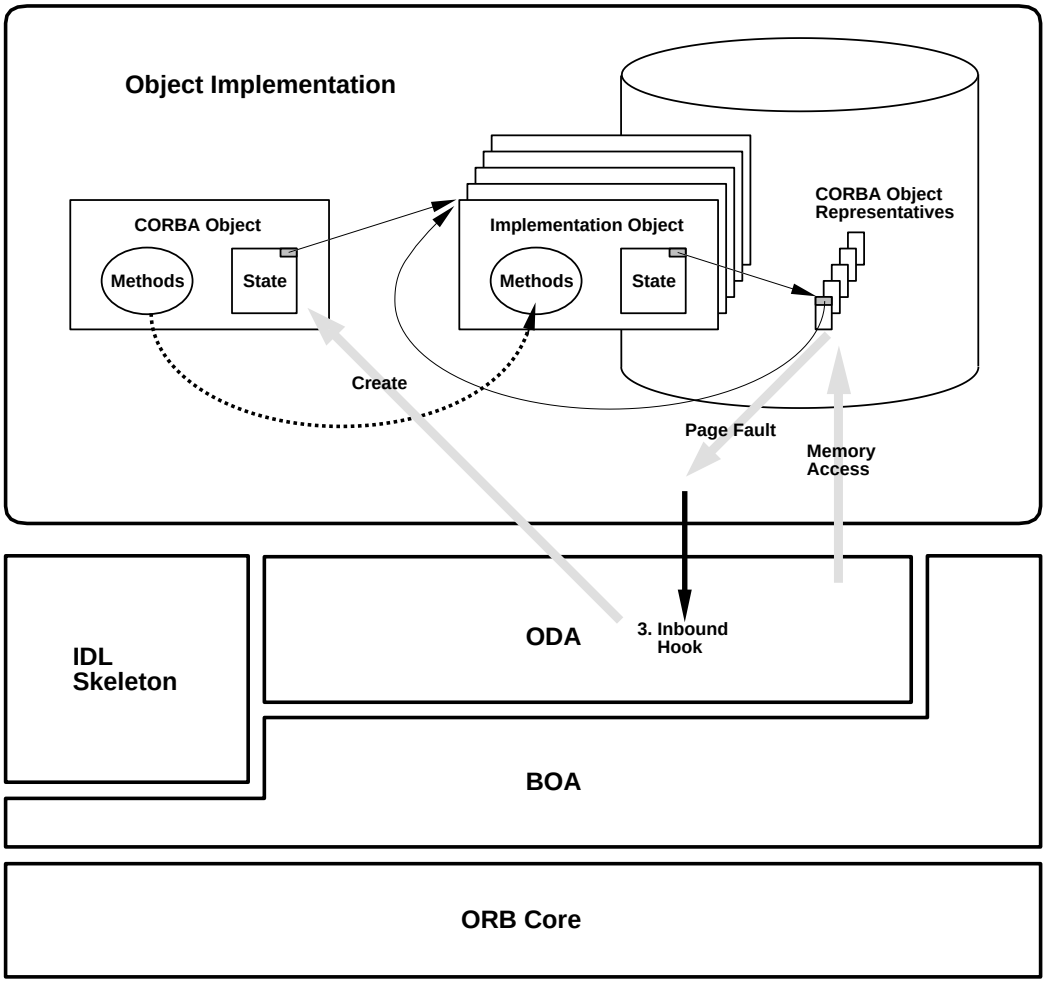



Figure 3.16: Incoming request handling in virtual persistence (part II). The memory access to the representative causes the instantiation of the target object.

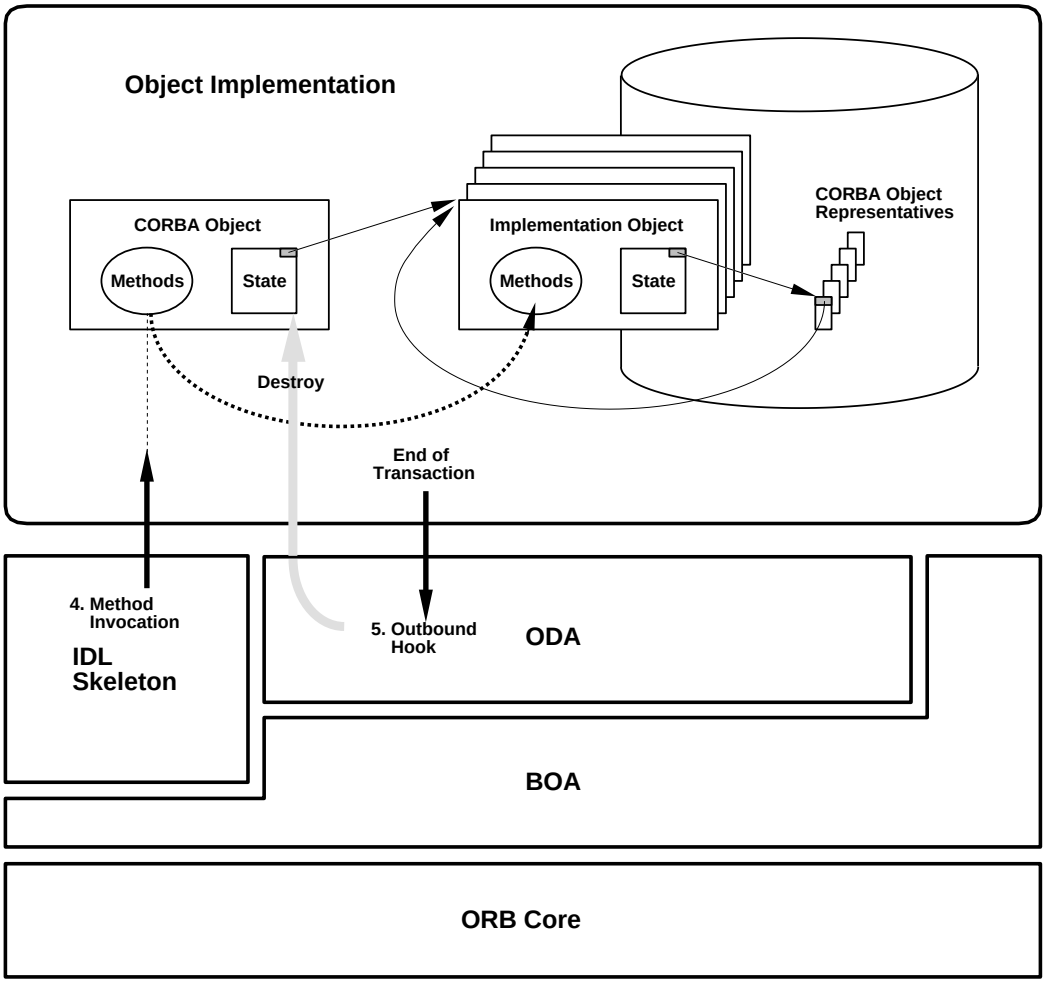



Figure 3.17: Incoming request handling in virtual persistence (part III). An up-call from the skeleton invokes the requested operation. At the end of the operation, an outbound hook deletes the target object.

```
// C++
class stub_A : public virtual A {
public:
    ...  // ORB-dependent member functions go here
    stub_A();
    virtual CORBA::Short op1();
    virtual void op2(CORBA::Long l);
    ...
    virtual ~stub_A();
};
```

Figure 3.18: IDL-generated stub class for interface A.

shows a stub class generated for the interface A by an IDL compiler that supports persistent references to remote CORBA objects. Note that this class has a single direct base, the interface class `A`, and that the inheritance is virtual. Also note that the functions corresponding to IDL operations are virtual member functions. These virtual member functions may be actually defined by the stub class, as in Figure 3.18, or may be simply inherited from the interface class; it does not matter for our scheme. Lastly, note that `stub_A` does not have any non-inherited data members.

A CORBA reference to a remote object of interface A is implemented in C++ as a pointer to a local object of class `stub_A`. The ORB automatically creates a stub object whenever a reference to a remote CORBA object enters the address space of a process, in any of the ways listed in 1.6. In particular, calls to `string_to_object` can create stub objects: when passed a string identifying a remote CORBA object that is not yet active in the current address space, `string_to_object` creates a stub for the object and returns its address. A stub object is deleted by `release`, when its reference count reaches zero.

To allow storing these CORBA references in a database, we introduce persistent *representatives* of stub objects. A reference to a remote CORBA object is then stored as a pointer to a representative of the corresponding stub object. Figure 3.19 shows the form of a stub representative. The `orb_ref` field is the only one actually written to the database. It points to a persistent string that identifies the remote CORBA object; this string was obtained through a call to `object_to_string`.

```
class stub_representative {
public:             // if the represented stub is a stub_A
    void* vbase;    //  this field points to its A part,
    void* vtbl;     //  this one to the virtual table of stub_A,
    char* orb_ref;  //  and this one to the corresponding
};                  //  CORBA reference, in string form
```

Figure 3.19: Representative of a remote CORBA object.

The ODA swizzles stub representatives at page fault time, just like it does with the tie representatives used for local objects. The only difference is that the represented objects — stub objects — are now instantiated via calls to `string_to_object`.

### The Overall Scheme

Unlike smart pointer-based persistence, virtual persistence is not a mere add-on to pseudopersistence: it introduces modifications in the pseudopersistence strategy described by items 1–6 in pages 67–69. Even so, the overall scheme for virtual persistence can still be characterized by that description, with the substitution of "pseudopersistent object" by "persistent CORBA object", plus the following additions and changes:

- Each persistent CORBA object must be registered with the ODA when first created[4]; the ODA then stores in the database a representative for the object.
- Each persistent implementation object holds a pointer to the representative of the persistent CORBA object it implements. The reason for storing these pointers within the implementation objects will be seen shortly.
- The ODA installs inbound and outbound hooks for each representative class. Instantiation and deletion of persistent CORBA objects, as well as swizzling of representatives, are performed by these hooks in the way already described.
- Item 3 in page 67 is modified: the object activation routine does not instantiate the transient ties that realize persistent CORBA objects anymore. These ob-

[4] "First", here, refers to the very first time the object was created, and not to the successive reinstantiations of the object by its inbound hook.

jects are now instantiated at page fault time; the instantiation of such an object is triggered by a mere access to the page that contains its representative. The object activation routine simply gets the address of the representative from the implementation object (this is why pointers to the corresponding representatives were stored within the implementation objects), considers this address as a reference to the represented object, and returns the result of the conversion of this reference to a reference to `CORBA::Object`[5].

- Items 5 and 6 in page 69 are removed: deactivation of persistent CORBA objects is now performed by outbound hooks. The reply handler and the "to release" list are not needed anymore.

Surprisingly enough, inbound/outbound hooks made the overall scheme look simpler.

## 3.7 ODA Support for Local Transactions

In the OMG architecture, distributed transactions are supported by the Object Transaction Service [39], which implements the two-phase commit protocol [12] and can be regarded as an object-oriented extension of X/Open DTP [60]. OTS implementations are expected during 1996, and will be probably based upon existing TP monitor products.

To be used with OTS and participate in distributed transactions, a DBMS must offer a resource manager interface, such as the set of XA primitives specified by X/Open DTP, or the `Resource` interface specified by OTS. The XA interface is supported by all major relational DBMSs, but most ODBMSs do not yet support a resource manager interface. For this reason, OTS will remain unavailable to ODBMS users in the near future.

Even without OTS, local (non-distributed) transactions still take place in an integrated ORB/ODBMS environment. Transactions local to a server are sufficient for many applications; this is the situation we consider here. In what follows, we discuss the actions an Object Database Adapter must take to support local transactions,

[5]Note that such conversion to `CORBA::Object` involves following the `vbase` pointer in the representative. The act of dereferencing this pointer triggers a page fault.

assuming the common case of an ODBMS that does not offer a resource manager interface.

Whatever approach is used to make CORBA objects persistent — simple persistence of implementation, pseudopersistence, or either form of full persistence — the methods that implement operations on these objects must be executed within a database transaction, because the ODBMS only allows access to persistent memory from within a transaction. One might think of asking the interface implementor to call `d_Transaction::begin` at the very start of each method, as well as `d_Transaction::commit` (or `abort`) at the very end of the method. This naive idea does not work, for the following reasons:

- Before a method starts executing, its target object must be activated. In order to instantiate a tie that realizes the target object, the activation routine needs to determine the address of the corresponding implementation object. This is a persistent memory address, which cannot be determined outside of a transaction. A similar argument applies to CORBA references passed as input parameters.

- References to persistent CORBA objects may be passed back to the client, either as output parameters or as the return value of the method. By the time these references are marshaled into a reply message, the objects they reference must be still active and accessible. This will not happen if the method itself calls `d_Transaction::commit`.

The ODA solves both problems by using request and reply handlers: a pre-marshal handler starts a transaction as soon as a request arrives, and a post-marshal handler ends the transaction just before the reply is sent. This solution encompasses each IDL operation by an individual transaction. This seems natural and is sufficient for many applications. In certain situations, however, it may be necessary to consider a sequence of IDL operations as a single database transaction.

### Multiple IDL Operations in a Single Transaction

From the point of view of the ODA, it is easy to accomplish this with request and reply handlers. What makes multi-operation transactions problematic is that the

execution of a sequence of operations defined as a transaction by a client may be interleaved, in the CORBA server, with the execution of operations requested by other clients. Transactions are typically implemented using database locks, usually *per-client* locks. Since a single database client — the CORBA server — is executing all operations, a whole set of interleaved operations would be then regarded by the ODBMS as a single transaction.

Orbix provides a server activation mode that solves this problem, at the cost of additional consumption of server machine resources: in this mode, called *per-client-process*, a separate CORBA server is automatically activated for each client process. This activation mode is orthogonal to the ones discussed in Section 1.4.1. It can be applied, for example, to a shared server: the resulting activation mode would be *shared-server-per-client-process*. In this composite activation mode, multiple active objects of a given interface are made accessible to a particular client process by a server. Requests from other client processes cause the activation of separate servers.

Such composite activation mode allows the safe implementation of multi-operation transactions: every client has a separate CORBA server, which in turn is granted its own database locks. Each sequence of operations defined as a transaction by a client is then regarded by the ODBMS as a separate transaction.

## 3.8 Relationship with POS

The Persistent Object Service (POS) [39] is one of the object services specified by the OMG. Designed to provide client control over object persistence, POS is geared towards objects that expose their persistence to CORBA clients.

A *Persistent Object* (PO) is defined as an object whose persistence is controlled externally by its clients. POS adopts a two-level store model: a Persistent Object has a dynamic state and a persistent state. The latter is typically kept in memory, and may not exist for the whole lifetime of the object. The PO interface defines the operations `connect`, `disconnect`, `store`, `restore`, and `delete`. Through these operations, clients can control the persistence of a PO. For instance, they can request an update of the persistent state with the contents of the dynamic state (`store`), or vice-versa (`restore`).

The PO interface is the client's view of this service. On the server side, POS defines

components used by PO implementations. A *Persistent Object Manager* (POM) provides a uniform interface to the underlying Persistent Data Services. A *Persistent Data Service* (PDS) provides a uniform interface for any combination of Datastore and Protocol. The persistent state of a PO is ultimately kept in a *Datastore*. Data is read and written to a Datastore through a *Protocol*, which depends both on the Datastore and on how data is moved into and out of (the dynamic state of) objects.

POS defines its lower level components in rather vague terms, in an attempt to encompass a large variety of storage services. Although no "standard" Protocol is specified, POS mentions three possible Protocols. One of them is the ODMG Protocol, used when the Datastore is an ODMG-compliant ODBMS. Rather than fully defining this Protocol, the POS specification discusses it very briefly, in a half-page section.

ORB/ODBMS integration and POS have opposite goals. Instead of exposing object persistence to clients, ORB/ODBMS integration provides object persistence in a way transparent to clients. Rather than adopting a two-level store model, ORB/ODBMS integration uses the ODBMS single-level store model and extends it to CORBA clients. Even so, it appears that POS still requires an Object Database Adapter, to deal with the case of an ODMG-compliant ODBMS as the Datastore and ODMG as the Protocol.

## 3.9 Related Work

When we started this work, ORB/ODBMS integration was a goal unanimously recognized as important, but not accomplished by any existing software. Since then, a number ORB and ODBMS vendors announced plans to integrate their products. Due to the commercial interests involved, however, it is unlikely that design and implementation information on these ORB/ODBMS integration efforts will be published.

Among these projects, Iona's Orbix+ObjectStore Adapter, OOSA ([19], [20]), is the one at the most advanced stage. OOSA was in beta test until very recently, and is being released as a product at the present time. Despite several vendor announcements, OOSA seems to be the only Object Database Adapter commercially available today.

### 3.9.1 Iona's Orbix-ObjectStore Adapter

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 object reference count is one of them.

OOSA actually stores CORBA objects in ObjectStore databases. A CORBA object, however, is not stored in their entirety: to avoid the performance penalty of having object reference counts in persistent memory, OOSA does not store the auxiliary object in the database. Instead, auxiliary objects are dynamically instantiated. When an auxiliary object is instantiated, the corresponding CORBA object is inserted into the per-process table of active objects maintained by Orbix.

OOSA seems to be still evolving with respect to the instantiation of auxiliary objects. In its alpha release, auxiliary objects were instantiated at page fault time, through the use of an ObjectStore inbound hook. They were deleted by an outbound hook.

The beta release of OOSA took a different approach: when an operation on a persistent CORBA object is invoked, OOSA-beta checks if the CORBA object has an auxiliary object or not, and instantiates the auxiliary object at this time. Auxiliary objects are still deleted by an outbound hook. It appears that this change was motivated by performance reasons, as it avoids the use of an inbound hook. But it introduced a problem: since the instantiation of an auxiliary object is followed by an assignment to persistent memory (the `CORBA::Object` data member that points to the auxiliary object), read-only transactions ceased to work[6].

OOSA-alpha implemented a variation of virtual persistence; OOSA-beta resembles smart pointer-based persistence. Neither OOSA release addressed the issue of storing CORBA references to remote objects.

[6]Read-only transactions worked in the alpha release of OOSA. In that release the critical assignment was performed by an inbound hook, using ObjectStore's "hidden write" function, which can only be called from an access hook.

### Comparison: OOSA-alpha and Virtual Persistence

OOSA-alpha uses ObjectStore access hooks in a way similar to virtual persistence. They differ with respect to what is stored in the database:

- Except for the associated auxiliary objects, OOSA-alpha makes full CORBA objects persistent. All data members inherited from `CORBA::Object` that do not live in an auxiliary object are actually stored. Database space is also taken by the pairs of `vbase` and `vtbl` pointers that appear for each interface class in the object's inheritance chain up to `CORBA::Object`.
- Virtual persistence uses the delegation approach to store only representatives with minimum size possible.

Because both approaches use ObjectStore access hooks, they should have similar performances. The advantage of OOSA-alpha is that it allows persistent CORBA objects to be implemented by inheritance or by delegation. Virtual persistence requires the use of delegation. The advantages of virtual persistence are:

- Minimal consumption of database space.
- No ORB-specific information is stored persistently. In principle, a database can be accessed by servers implemented with different ORBs. With OOSA-alpha this is not possible. Even in the case of Orbix, if object layout changes in a future release, all existing databases will have to go through a schema evolution process.

### Comparison: OOSA-beta and Smart Pointer-Based Persistence

The lazy approach to instantiation of auxiliary objects taken by OOSA-beta resembles smart pointer-based persistence, which does lazy instantiation of the CORBA objects identified by ODA references. We discussed smart pointer-based persistence in the context of an ODMG-compliant ODBMS, and therefore used the an ODMG function (`d_activate`) to ensure that a smart pointer has its "plain pointer" data member set to null upon entry into the application cache. OOSA-beta does something equivalent:

through the use of an ObjectStore hook[7] (an outbound hook), it ensures that a persistent CORBA object has its auxiliary object pointer set to null upon exit from the application cache. The relative advantages and disadvantages of OOSA-beta and smart pointer-based persistence parallel the ones of OOSA-alpha and virtual persistence.

[7]Note that ObjectStore inbound and outbound hooks are analogous to the ODMG `d_activate` and `d_deactivate` functions.

# Chapter 4

# The Sunrise ODA

The unavailability of a CORBA-based environment with full support for object persistence motivated the development of the Sunrise ODA. We designed and implemented this Object Database Adapter as part of our work on the Sunrise Project, a National Information Infrastructure effort undertaken by the LANL Advanced Computing Laboratory.

The Sunrise ODA was initially targeted at Iona's ORB, Orbix, and Object Design's ODBMS, ObjectStore. The reasons for choosing Orbix were:

- its close compliance with the CORBA standard, including full support to object activation;
- its support to the delegation approach to interface implementation;
- its comprehensive set of request and reply handlers;
- its availability on a wide range of operating systems and hardware platforms.

The main reason for choosing ObjectStore was its importance within the ODBMS market. The ObjectStore virtual memory approach actually made ORB/ODBMS integration a more challenging problem, as well as a more interesting one.

We later ported the Sunrise ODA to a relational engine, mSQL [13], which was accessed in an object-oriented fashion (as an ODBMS), through a simple smart pointer-based object-relational mediator. Subsequently we ported it to a second ORB, Postmodern Computing's ORBeline. Currently there are releases of this ODA for:

- Orbix and ObjectStore;
- Orbix, mSQL, and our own object-relational mediator;
- ORBeline and ObjectStore.

The Sunrise ODA is implemented as a fully tested library. Its "standard version" (see Section 4.1) is portable to whatever environments for which the supported ORBs and ODBMSs are available, and is currently being used on Solaris 2.x and IRIX 5.x.

## 4.1 Persistence Approaches Supported

The "standard version" of the Sunrise ODA provides pseudopersistent C++ CORBA objects (see Section 3.5). An extended version supports also virtual persistence, and lets the interface implementor freely mix these approaches within a server. We wrote the extended version with the goal of allowing a particular CORBA server to achieve the tradeoff between persistence and performance levels that best suits the objects it implements. So far this extended version integrates Orbix and ObjectStore, runs on Solaris 2.x, and requires the use of a particular C++ compiler, the SPARCompiler 4.0.1[1]

Even in its "standard version", the Sunrise ODA also supports mere persistence of implementation objects (see Section 3.3.2). This option is present mostly for historical reasons; it was the first approach we implemented. It is still has some utility today, for *top-level objects*. Among the many objects implemented by a CORBA server, there is usually a "top-level" one, which represents the server itself. The top-level object of a server is known to its clients, which use it to start interacting with the server. A top-level object uses simple persistence of implementation and remains active as long as its server is active, because under this approach it can be assigned an `id` (*marker*, in Orbix terminology)[2] arbitrarily chosen by its server. This allows a server to choose

[1]The extended ODA is not portable: besides being targeted at a virtual memory-based ODBMS, virtual persistence uses compiler dependent information on the memory layout of C++ objects.

[2]ODA-assigned object reference `id`s are used in the pseudopersistence and virtual persistence approaches.

an `id` known to its clients, which in turn use this `id` to locate the server's top-level object[3].

## 4.2 Dependency Upon ORB and ODBMS Features

Among the Orbix and ORBeline features not generally available in all ORBs, the following ones are used by the ODA:

- full support to object activation[4], through user-defined loader (Orbix) or activator (ORBeline) classes;
- support to the delegation approach to interface implementation;
- incoming request pre-marshal and outgoing reply post-marshal handlers.

The ODBMS features used by the ODA depend on its version ("standard" or extended), that is, on the approaches to persistence of CORBA objects it supports.

Pseudopersistence uses only object identity (ODBMS object references), plus the ability to convert ODBMS references to strings and vice-versa. Because these are fairly common ODBMS features, other ODBMSs can be employed instead of ObjectStore. All that is needed is a simple adaptation of the ODA code. With the use of an object-relational mediator, even a relational DBMS can be employed. In this case, primary keys play the role of ODBMS references. The mSQL release of ODA demonstrates the applicability of the pseudopersistence scheme to relational DBMSs.

Virtual persistence, however, requires also a virtual memory-based ODBMS that offers page fault hooks (see page 79). In the case of a smart pointer-based ODBMS, the scheme discussed in Section 3.6.1 would have to be implemented instead.

---

[3]Locating servers by their `id`s is an interim solution. Such practice, supported by Orbix in substitution to an actual *name service*, is not in the spirit of the OMG architecture: according to CORBA, object `id`s should only be meaningful to the object implementations themselves. This use of `id`s will be dropped when Iona's implementation of the CORBA Name Service becomes available.

[4]Although object activation is in the CORBA standard, we have included it in this list because it is neither fully defined by CORBA nor supported by all ORB implementations.

## 4.3 ODA Utilization by TeleMed

The Sunrise ODA is currently providing persistence of CORBA objects to TeleMed, a distributed, object-oriented, CORBA-based telemedicine system, a joint project of the Los Alamos National Laboratory and the National Jewish Center for Immunology and Respiratory Medicine in Denver, CO. This application, a component of the LANL Sunrise project, drove the development of the ODA and supplied us with a realistic test environment for the ORB/ODBMS integration approaches discussed in this dissertation.

TeleMed objects are stored in ObjectStore databases and accessed via Orbix, through IDL-defined interfaces. Except for top-level objects, for which persistence of implementation is employed, CORBA objects use the pseudopersistence approach. Due to both performance (see Chapter 5) and DBMS independence concerns, we decided not to use virtual persistence at the present time.

The relationships between CORBA objects in a TeleMed database are therefore represented by links between implementation objects; ObjectStore allows the use of plain C++ pointers to implement these links. The relationships between CORBA objects in different TeleMed databases are represented by CORBA references converted to strings.

This scheme provides maximum efficiency and a good level of DBMS independency, at the cost of some increase in the complexity of the server code. As an example of this increase in code complexity, consider a situation in which it would be desirable to have a homogeneous list of CORBA references, whose elements may refer either to local or to remote objects. Instead of a single and homogeneous list, two separate lists must be introduced: one with pointers to local implementation objects, other with stringfied CORBA references to remote objects.

# Chapter 5

# The Performance of ODA Approaches

We implemented an "ODA benchmark", a simple test suite to evaluate the approaches to ORB/ODBMS integration discussed in this dissertation. The actual performance of each approach for a particular application, of course, will depend on the characteristics of the application. Designing a benchmark to estimate how well an Object Database Adapter does on a typical application was not one of our aims: distributed object applications are just starting to appear, and there is no "typical" one at the present time.

## 5.1 The Test Environment

Both the database server and the CORBA server were placed on the same machine, a Sun SPARCstation 10 with 128 megabytes of memory, running Solaris 2.5. For the CORBA clients, we used another SPARCstation 10, this one running Solaris 2.3.

The tests were performed in a real-world environment: other users had access to the client and server machines, which were on an Ethernet LAN shared by about 60 hosts. Although we did all measurements during hours of low machine load and light network traffic, these factors may still have had some effect on our results. A more rigorous study would require dedicated machines on an isolated piece of Ethernet.

Orbix 1.3.4 and ObjectStore 4.0.0.B were used for all tests. Persistent objects were kept in ObjectStore file databases, in a 2 gigabyte SCSI-2 drive (model Seagate

Hawk) connected to the server machine. We used the default size of 8 megabytes for the ObjectStore client cache, and set to 8192 entries the size of the table of active objects maintained by Orbix, a hash table with overflow chains. Test programs were compiled with the SPARCompiler C++ 4.0.1, and used either the extended version of the Sunrise ODA (simply referred to as ODA in the rest of this chapter), or the beta release of Iona's OOSA.

## 5.2 ORB and ODBMS Performance Figures

We obtained these numbers in the environment described in Section 5.1.

### 5.2.1 Remote Method Invocation (Orbix)

A CORBA method invocation takes approximately 2.2 milliseconds. This is the round-trip time of an operation that takes no parameters, performs no work, and returns no results. Table 5.1 shows the round-trip times of a null operation that receives an octet sequence of length $\ell$ as input parameter, for various values of $\ell$.

| $\ell$ | time |
|---|---|
| 256 | 2.5 |
| 1024 | 3.2 |
| 4096 | 5.8 |
| 16384 | 18.4 |
| 65536 | 67.7 |

Table 5.1: Null operation with octet sequence of length $\ell$ as input parameter. (Times in milliseconds.)

### 5.2.2 Persistent Memory Access (ObjectStore)

Any accesses to persistent memory must be performed within a transaction. After a transaction was started, objects already cached by the database client are accessed at memory speed. The transaction cost, however, is significant: it takes 870 microseconds to start and commit a read-only transaction that accesses no data and executes no work.

Inbound and outbound access hooks introduce additional and much greater costs. Tables 5.2 and 5.3 show the results of an experiment we performed to evaluate these costs. A database client traverses a persistent list whose elements have two data members, `name` and `data`, both of type `char*`. The `name` fields are set to `"Element(0)"`, `"Element(1)"`, and so on. The `data` fields are set to strings of spaces, all with the same length. The database client does a full traversal of the list, searching for a name not assigned to any element. The results that follow correspond to a list with 2048 elements and to a data field length of 4096.

Table 5.2 presents "cold times", obtained when the client cache is empty. Table 5.3 shows the "warm times" obtained in a second traversal, when most of the persistent data is already in the 8 megabyte client cache. Each list traversal was performed within a single transaction. Traversal times include the corresponding start transaction calls; commit transaction calls were timed separately. We measured cold and warm times in four situations:

| | traversal time | | | | commit time | | | | **total time** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AH | real | cpu | usr | sys | real | cpu | usr | sys | **real** | cpu | usr | sys |
| — | 16.79 | 7.74 | 4.69 | 3.05 | 0.07 | 0.07 | 0.02 | 0.05 | **17.12** | 7.97 | 4.84 | 3.13 |
| I | 49.32 | 40.26 | 37.12 | 3.14 | 0.09 | 0.09 | 0.03 | 0.06 | **49.69** | 40.51 | 37.28 | 3.23 |
| O | 50.63 | 41.36 | 38.08 | 3.28 | 33.40 | 33.19 | 32.10 | 1.09 | **84.30** | 74.72 | 70.34 | 4.38 |
| I,O | 51.52 | 42.29 | 39.11 | 3.18 | 33.44 | 33.23 | 32.25 | 0.98 | **85.25** | 75.65 | 71.46 | 4.19 |

Table 5.2: Cold traversal of persistent list with 2048 elements. Each element of the list has a `data` field of length 4096. (AH = access hooks installed; I = inbound hook; O = outbound hook; times in seconds.)

| | traversal time | | | | commit time | | | | **total time** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| AH | real | cpu | usr | sys | real | cpu | usr | sys | **real** | cpu | usr | sys |
| — | 2.34 | 2.31 | 1.15 | 1.16 | 0.06 | 0.06 | 0.02 | 0.04 | **2.41** | 2.38 | 1.18 | 1.20 |
| I | 36.10 | 35.83 | 34.39 | 1.44 | 0.08 | 0.08 | 0.03 | 0.05 | **36.22** | 35.95 | 34.46 | 1.49 |
| O | 38.56 | 37.53 | 35.95 | 1.58 | 33.24 | 33.07 | 31.90 | 1.17 | **71.87** | 70.64 | 67.89 | 2.75 |
| I,O | 39.76 | 38.33 | 36.70 | 1.63 | 33.33 | 33.11 | 32.08 | 1.03 | **73.16** | 71.48 | 68.82 | 2.66 |

Table 5.3: Warm traversal of persistent list with 2048 elements. Each element of the list has a `data` field of length 4096. (AH = access hooks installed; I = inbound hook; O = outbound hook; times in seconds.)

- without any access hooks installed;
- with an inbound hook that performs no work;
- with an outbound hook that performs no work;
- with both hooks installed.

Note the warm time degradation by a factor of 15, when an inbound hook is used, and by a factor of 30, when an outbound hook is used.

ObjectStore usually does a good job on caching persistent objects in the database client. Unfortunately its caching mechanism ceases to work in the presence of an access hook; the vendor informed us that access hooks force the client cache to be flushed out and reloaded again[1]. We regard this as an ObjectStore problem: flushing and reloading the cache is an action too drastic and not really necessary. Some access hook overhead was expected, but we believe it could be substantially smaller.

## 5.3 Benchmark Description

Our focus in on the performance of database accesses by CORBA clients, in an integrated ORB/ODBMS environment. We are interested in the effects of ODA design decisions on the efficiency of the integrated environment, not in database performance by itself. Therefore our synthetic ODA benchmark is much simpler than an ODBMS benchmark, such as [5], which addresses a whole range of database issues: indexing, joins, cached updates, deletes, etc. Though still important in a CORBA environment, these issues are orthogonal to the way ORB/ODBMS integration is realized.

### 5.3.1 IDL Interfaces

In an attempt to mimic object relationships found in many applications, the ODA benchmark is based upon a tree of CORBA objects. Figure 5.1 shows the IDL interface of a `Node` object.

[1]This information is consistent with the data in Tables 5.2 and 5.3. The inbound hook overhead on the warm traversal time is of approximately twice the cold traversal time. The outbound hook overhead is roughly the same, but shows up in both the traversal and in the commit time.

```
interface Node;
typedef sequence<Node> NodeList;
typedef sequence<string> StringList;

interface Node {
    readonly attribute string name;
    attribute string data;
    readonly attribute NodeList children;
    readonly attribute long descendant_count;
    readonly attribute StringList descendant_names;
    void add_child(in Node new_child);
};
```

Figure 5.1: The `Node` interface.

Each `Node` has a `name`, some `data`, and `children`, a list of child nodes. The `data` attribute was included to give CORBA clients control over the granularity of `Node` objects. Although `descendant_count` and `descendant_names` are `Node` attributes, they are not directly stored in the database. Instead, they are computed by `Node` implementations, through a recursive traversal the sub-tree rooted at the node. The `add_child` operation inserts its input parameter, an existing node, into the child list of the target node. Before being passed to `add_child`, a free `Node` has to be created. This is the purpose of the `NodeFactory` interface in Figure 5.2.

```
interface NodeFactory {
    Node create(in string name, in string data);
    Node create_with_default_data(in string name,
                                  in unsigned long data_length);
};
```

Figure 5.2: The `NodeFactory` interface.

The "top-level" object in a server is a `Tree`; Figure 5.3 shows its interface. Not surprisingly, a `Tree` has a `root_node`. Because it represents a CORBA server, the `Tree` also has two "server attributes", a `NodeFactory` and a `TransactionControl` object; the latter supports the interface shown in Figure 5.4.

The `TransactionControl` interface supports multi-operation transactions. By

```
interface Tree {
    readonly attribute Node root_node;
    readonly attribute NodeFactory node_factory;
    readonly attribute TransactionControl transaction_control;
};
```

Figure 5.3: The `Tree` interface.

```
interface TransactionControl {
    void begin_readonly_transaction();
    void begin_update_transaction();
    void commit_transaction();
};
```

Figure 5.4: The `TransactionControl` interface.

default, each operation on a `Node`, `NodeFactory`, or `Tree` corresponds to a database transaction. To start and commit multi-operation transactions, CORBA clients issue requests to the `TransactionControl` object of a `Tree`.

There is no operation to create a `Tree`. An initial `Tree` object is automatically created by the first server run, and its state is stored in a database. Subsequent runs of the server use the last `Tree` state stored. The initial `Tree` has no other nodes besides its `root_node`, whose `name` and `data` attributes are initialized to the strings `"Node(0)"` and `"root node data"`, respectively.

### 5.3.2 The Servers

We used a set of servers to evaluate the different ORB/ODBMS integration approaches. These servers implement the same IDL interfaces, but differ on how they make `Node` objects persistent.

**Pseudopersistence** was represented by server ODA-pp, which uses ODA and implements pseudopersistent `Node` objects.

**Smart pointer-based persistence** was represented by two servers:

- A server that uses the beta release of OOSA and makes `Node` objects persistent through this adapter's approach, which resembles smart pointer-based persistence. We call this server OOSA-$\beta$.
- A server that uses ODA and simulates smart pointer-based persistence of `Node` objects. We call this server ODA-sp.

None of these servers actually employs the smart pointer approach described in Section 3.6.1. Server OOSA-$\beta$ uses an approach similar to smart pointer-based persistence, but implemented without adequate ODBMS support. Server ODA-sp simulates the behavior that smart pointer-based persistence would exhibit, given adequate ODBMS support. The meaning of *adequate ODBMS support to smart pointer-based persistence* is defined below.

**Virtual persistence** was represented by server ODA-vp, which uses ODA and implements virtually persistent `Node` objects.

Each of these servers employs the shared server activation mode (see Section 1.4.1) and implements a whole tree, which comprises the top-level `Tree` object, the root `Node` and its descendants, and the `Tree`'s `NodeFactory` and `TransactionControl` objects. `Tree` and `Node` objects are made persistent by the CORBA server. `NodeFactory` and `TransactionControl` objects have no persistent state. They are instantiated when the server initializes itself, and remain active until the server exits.

### ODBMS Support for Smart Pointer-Based Persistence

Since smart pointer-based persistence cannot be efficiently implemented without adequate ODBMS support, we used a server that simulates this approach. In order to support smart pointer-based persistence in a satisfactory way, an ODBMS must offer:

1. an efficient way of ensuring that a persistent smart pointer has its "plain pointer" data member set to null before the smart pointer is first dereferenced in a transaction[2];

[2]As discussed in Section 3.6.1, this can be accomplished by using the function `d_activate` to ensure that the pointer is set to null upon entry into the application cache. Another possibility is using `d_deactivate` to ensure that the pointer is set to null upon exit from the application cache.

2. an efficient way of updating this data member *in the application cache only*, without write locking the smart pointer object or marking it as modified.

In each of these requirements, the adjective "efficient" means that the corresponding action is performed with negligible overhead. Both requirements are satisfied by an ODMG-compliant and smart pointer-based ODBMS:

- A persistent smart pointer class can accomplish 1 by redefining the virtual member function `d_activate`. Since the redefined `d_activate` simply sets a pointer to null, its overhead is negligible. Note that no extra function calls are performed due to the redefinition of this virtual function[3].

- It can accomplish 2 through an assignment to its "plain pointer" data member, with no corresponding `mark_modified` call. This assignment can be performed even by a read-only transaction[4], without a write-lock to the smart pointer object.

Most existing ODBMSs are smart pointer-based, and are expected to become ODMG-compliant in the near future. When this happens, the majority of ODBMSs will offer adequate support to smart pointer-based persistence.

ObjectStore meets neither of the requirements for smart pointer-based persistence. Its access hooks provide the functionality of requirement 1, but at a very high cost (see Section 5.2.2). Its "hidden write" function provides the functionality of requirement 2, but can be called only from an access hook.

The results we obtained with server ODA-sp reflect the *hypothetical* performance of smart pointer-based persistence on ObjectStore, i.e., the performance that would be achieved *if this ODBMS satisfied requirements 1 and 2*. Similar results would be obtained with an actual implementation of smart pointer-based persistence on

---

Alternatively, the ODBMS could allow the programmer to specify that a data member of a given class is a transient pointer, which the ODBMS would automatically set to null upon entry into the cache.

[3]The inherited function `d_Object::d_activate` would be called otherwise.

[4]A *smart pointer-based* ODBMS rejects `mark_modified` calls within read-only transactions, but has no way of disallowing an assignment to a data member of a persistent object in the application cache.

an ODBMS comparable to ObjectStore in performance, but with adequate support to this approach. The results we obtained with server OOSA-$\beta$ reflect the *actual* performance of smart pointer-based persistence on ObjectStore.

### Server ODA-pp

Server ODA-pp implements pseudopersistent `Node` objects. Relationships that logically lead to `Node`s are physically realized by persistent pointers[5] to the corresponding implementation objects, of class `Node_impl`:

- The `Tree` attribute `root_node` is implemented by a persistent pointer to a `Node_impl` object.
- Lists of child nodes are stored as lists of pointers to `Node_impl` objects.

By following such a relationship, the server actually reaches a `Node_impl` object, which it uses to directly invoke any needed operations on the node.

To make the last point clear, consider the recursive computation of the `Node` attribute `descendant_count`; the IDL-generated class `Node` has an accessor function that returns this attribute. The IDL-generated accessor simply invokes an implementation-provided accessor, a member function of class `Node_impl`. The recursive computation is performed by `Node_impl::descendant_count()`, which traverses the list of child nodes and *directly calls itself* on each child node.

For the top-level `Tree` object, server ODA-pp uses simple persistence of implementation (see Section 3.3.2). When the server initializes itself, it instantiates a `Tree` object that acts as an IDL shell to the persistent `Tree_impl` object. This IDL shell remains active as long as the server is active.

### Server ODA-sp

Since ODA does not support smart pointer-based persistence, server ODA-sp simulates this approach. Objects are stored exactly as in server ODA-pp: persistence of implementation is used for the top-level `Tree` object, and `Node` objects are pseudo-

[5]Since we are using a virtual memory-based ODBMS, plain C++ pointers can be made persistent.

persistent. Smart pointer-based persistence is simulated by additional actions of the server; these actions take place when database relationships are traversed.

Whenever it follows a database relationship leading to a `Node_impl` object, server ODA-sp acts in the following way:

- Immediately after reaching the `Node_impl` object, the server obtains (by calling the ODA) a CORBA reference to the corresponding `Node` object.
- The server refrains from directly using the `Node_impl` object to invoke operations on the node. Instead, it invokes any needed operations through the CORBA reference to the corresponding `Node`.

Consider again the computation of `descendant_count`. Now the implementation-provided accessor `Node_impl::descendant_count()` calls itself indirectly: it obtains CORBA references to the child nodes, and uses these references to invoke the IDL-generated accessor `Node::descendant_count()` on each child node.

Server ODA-sp does not simulate smart pointer-based persistence completely, but its performance is an upper-bound for the performance of smart pointer-based persistence. In the case of a transaction that reaches a `Node_impl` object more than once, server ODA-sp calls the ODA to obtain the corresponding CORBA reference as many times as the implementation object is reached. Only the first call instantiates a `Tree` object (if the object is not active yet), but each remaining call still performs a lookup on the table of active pseudopersistent objects maintained by the ODA.

An actual implementation of smart pointer-based persistence would avoid these table lookups. A data member of the smart pointer class `ODA_Ref<Node>` would keep the CORBA reference obtained when a smart pointer is first dereferenced within a transaction. Subsequent traversals of this smart pointer in the same transaction would reuse that CORBA reference. On an ODBMS with adequate support for smart pointer-based persistence, this approach would perform no worse than server ODA-sp.

### Server ODA-vp

Server ODA-vp implements virtually persistent `Node` objects. Relationships that logically lead to `Node`s are physically realized by persistent pointers to `Node` representatives:

- The `Tree` attribute `root_node` is implemented by a persistent pointer to a `Node` representative.
- Lists of child nodes are stored as lists of pointers to `Node` representatives.

By following such a relationship, the server reaches a `Node` representative. Because the representative is swizzled into a `Node` at page fault time, it appears as such to the server. For the top-level `Tree` object, server ODA-vp uses simple persistence of implementation (see Section 3.3.2), like the previous servers.

The recursive computation of `descendant_count` now happens as follows. The implementation-provided accessor `Node_impl::descendant_count()` perceives the list of child nodes as a list of CORBA references to `Node`s. It traverses this list, calling the IDL-generated accessor `Node::descendant_count()` on each element.

The overhead of ObjectStore access hooks (see Section 5.2.2) severely impairs the results obtained with server ODA-vp. It makes these results less interesting, as they show virtual persistence performing much worse than it could perform with adequate ODBMS support. To reduce this effect, we introduced the following modification on ODA's implementation of virtual persistence.

Instead of employing outbound hooks to deactivate persistent CORBA objects, as described in Section 3.6.2, the ODA uses an Orbix reply handler. Deactivation of persistent CORBA objects is performed as in the pseudopersistence approach (see Section 3.5). With this modification, the ODA avoids the outbound hook overhead, incurring the cost of inbound hooks only.

#### Server OOSA-$\beta$

Server OOSA-$\beta$ makes the `Tree` and its `Node`s persistent, as described in Section 3.9.1. Relationships that lead to `Node`s are realized by persistent pointers to `Node` objects. The recursive computation of `descendant_count` happens as in server ODA-vp. Since OOSA does not support persistence of implementation objects, the top-level `Tree` object is accessed by clients through its OOSA-assigned `id` (see Section 4.1).

### 5.3.3 The Clients

The ODA benchmark implements a set of CORBA clients:

- clients that populate a brand new tree in breadth or depth order, creating new nodes and adding them as children of existing nodes;
- a client that obtains the number of nodes of a tree by calling `descendant_count` on the root node;
- a client that obtains the number of nodes of a tree by recursively visiting all its nodes;
- a client that searches for a node with a given name.

The same set of clients is used with all servers. This is possible because these clients interact with any server exclusively through the IDL interfaces presented in Section 5.3.1, which are common to all servers.

## 5.4 Results

In the tests we employed complete quad-trees. Each non-leaf node of such a tree has degree 4, and the leaf nodes are all at the same level. Table 5.4 shows the number of nodes of complete quad-trees of different heights.

| tree height | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| number of nodes | 1 | 5 | 21 | 85 | 341 | 1365 | 5461 |

Table 5.4: Number of nodes of complete quad-trees.

The number of objects in the database (the height of the tree) varied across test runs, and so did the granularity of these objects (the node data length). We ran all tests with tree heights from 1 to 5 and with data lengths from 64 to 16384.

In what follows:

- response times are expressed in seconds;
- $h$ denotes the height of a tree;
- all `Nodes` added to a `Tree` object have `data` attributes with the same length, which we denote by $\ell$;

- when populating a quad-tree object, `name` attributes are assigned to its `Node`s in the following way:
  - the root received the name `"Node(0)"` when the `Tree` was created;
  - its child nodes are named `"Node(0,0)"`, `"Node(0,1)"`, `"Node(0,2)"`, and `"Node(0,3)"`;
  - the children of a node named `"Node(`$s$`)"`, where $s$ is any string, are named `"Node(`$s$`,0)"`, `"Node(`$s$`,1)"`, `"Node(`$s$`,2)"`, and `"Node(`$s$`,3)"`.

### 5.4.1 Database Creation

**Test 1:** *Populate a tree in level order, in a single transaction, through a series of calls to* `NodeFactory::create_with_default_data` *and* `Node::add_child`.

Since this test involves a relatively long sequence of operations on multiple objects, we have used it to check the effectiveness of object caching on the pseudopersistence approach. Performing the whole sequence of operations within a single transaction avoids repeated transaction overheads, which could mask the benefits of caching.

We ran Test 1 on server ODA-pp, with the size of the cache of active pseudopersistent objects set to 8191 entries (full caching), and with this size set to zero (no caching). Table 5.5 and Figure 5.5 (left) present the response times for $\ell = 4096$ and various values of $h$. Table 5.6, Figure 5.5 (right), and Figure 5.6 show the numbers we obtained by varying $\ell$.

| $h$ | ODA-pp with caching | ODA-pp without caching |
|---|---|---|
| 1 | 0.31 | 0.35 |
| 2 | 0.67 | 0.81 |
| 3 | 2.18 | 2.73 |
| 4 | 7.30 | 10.21 |
| 5 | 28.85 | 41.62 |
| 6 | 148.70 | 216.03 |

Table 5.5: Test 1 — ODA-pp with and without caching, $\ell = 4096$.

| $\ell$ | $h = 4$ | | $h = 5$ | | $h = 6$ | |
|---|---|---|---|---|---|---|
| | ODA-pp with caching | ODA-pp without caching | ODA-pp with caching | ODA-pp without caching | ODA-pp with caching | ODA-pp without caching |
| 64 | 4.24 | 7.18 | 17.15 | 29.32 | 68.10 | 140.99 |
| 256 | 4.55 | 7.24 | 18.77 | 30.09 | 68.54 | 142.15 |
| 1024 | 5.10 | 7.96 | 20.58 | 32.50 | 81.54 | 147.47 |
| 4096 | 7.30 | 10.21 | 28.85 | 41.62 | 148.70 | 216.03 |
| 16384 | 19.38 | 22.23 | 105.04 | 113.34 | — | — |

Table 5.6: Test 1 — ODA-pp with and without caching, $h = 4, 5, 6$.

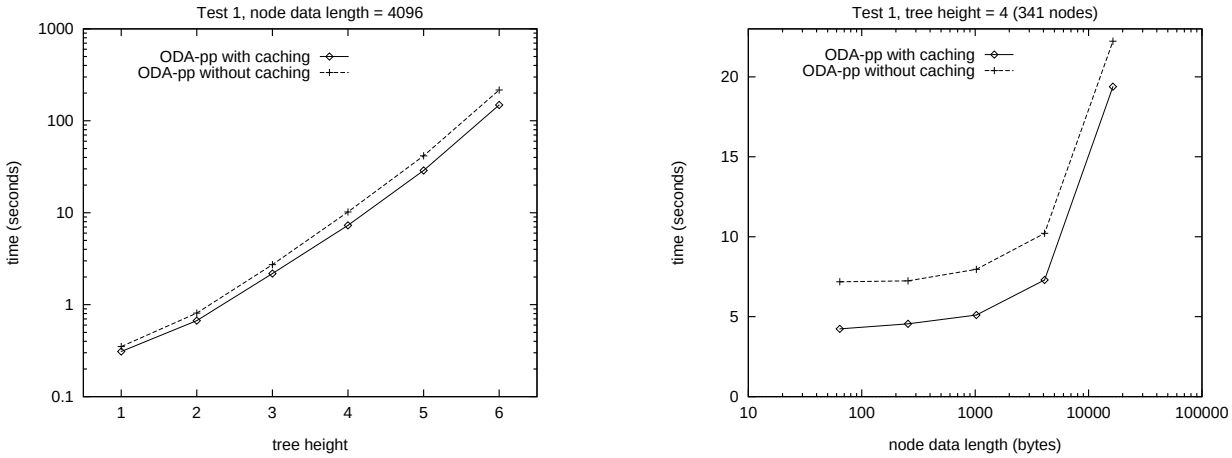



Figure 5.5: Test1 — ODA-pp with and without caching, $\ell = 4096$ (left), $h = 4$ (right).

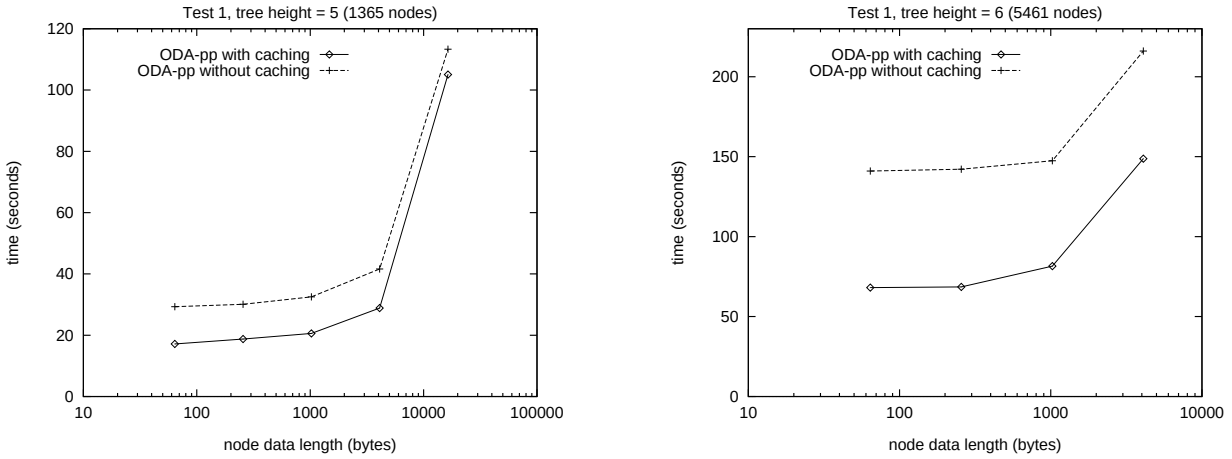



Figure 5.6: Test1 — ODA-pp with and without caching, $h = 5$ (left), $h = 6$ (right).

Figure 5.5 and 5.6 clearly show the positive effect of caching on pseudopersistence. In the remaining tests, servers ODA-pp and ODA-sp employ a cache of active pseudopersistent objects, with space for 8191 entries.

The results of Test 1 on all servers will be presented along with those of Test 2.

**Test 2:** *Populate a quad-tree in level order, in a single transaction, through a series of calls to* `NodeFactory::create` *and* `Node::add_child`.

This test only differs from Test 1 in that the `data` contents of the nodes are now transmitted from the client. Tables 5.7–5.9 and Figures 5.7–5.9 show the results of Tests 1 and 2 for all servers. As expected, response times for Test 2 are greater. These numbers also show the four servers divided in two groups, according to their performance: not surprisingly, the ones that do not use ObjectStore access hooks (ODA-pp and ODA-sp) are the best performers.

| $h$ | Test 1 ($\ell = 4096$) | | | | Test 2 ($\ell = 4096$) | | | |
|---|---|---|---|---|---|---|---|---|
| | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 1 | 0.30 | 0.32 | 0.38 | 0.38 | 0.33 | 0.38 | 0.39 | 0.39 |
| 2 | 0.65 | 0.71 | 1.05 | 1.02 | 0.73 | 0.75 | 1.17 | 1.10 |
| 3 | 2.11 | 2.07 | 3.62 | 3.84 | 3.74 | 2.46 | 3.89 | 3.98 |
| 4 | 7.55 | 7.10 | 13.32 | 14.35 | 12.30 | 8.38 | 14.39 | 15.03 |
| 5 | 29.44 | 30.56 | 51.81 | 55.03 | 34.45 | 34.26 | 56.61 | 60.91 |

Table 5.7: Tests 1 and 2, $\ell = 4096$.

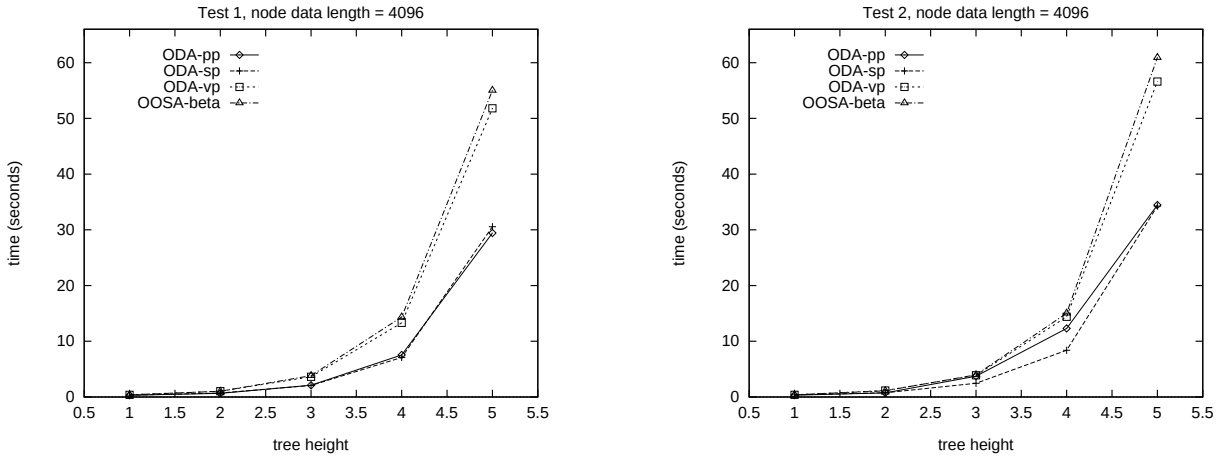



Figure 5.7: Tests 1 (left) and 2 (right), $\ell = 4096$.

| | Test 1 ($h = 4$) | | | | Test 2 ($h = 4$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 4.44 | 4.42 | 4.38 | 5.08 | 4.43 | 4.51 | 4.40 | 5.04 |
| 256 | 4.68 | 4.59 | 4.73 | 5.49 | 4.70 | 4.75 | 4.86 | 5.56 |
| 1024 | 5.53 | 5.36 | 6.19 | 7.11 | 5.67 | 5.62 | 6.58 | 7.28 |
| 4096 | 7.55 | 7.10 | 13.32 | 14.35 | 12.30 | 8.38 | 14.39 | 15.03 |
| 16384 | 18.00 | 18.46 | 76.52 | 77.34 | 31.79 | 24.36 | 82.39 | 83.56 |

Table 5.8: Tests 1 and 2, $h = 4$.

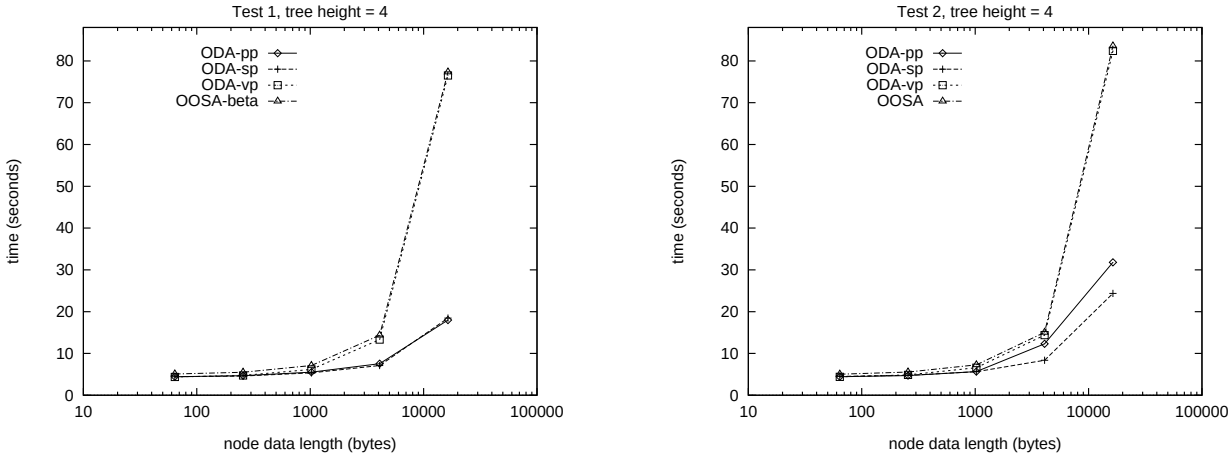



Figure 5.8: Tests 1 (left) and 2 (right), $h = 4$.

| | Test 1 ($h = 5$) | | | | Test 2 ($h = 5$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 17.01 | 17.09 | 16.42 | 19.64 | 17.33 | 17.61 | 16.79 | 19.64 |
| 256 | 18.07 | 17.85 | 18.57 | 20.99 | 17.99 | 18.59 | 18.30 | 21.41 |
| 1024 | 20.13 | 19.90 | 23.39 | 26.64 | 21.52 | 21.16 | 25.02 | 28.09 |
| 4096 | 29.44 | 30.56 | 51.81 | 55.03 | 34.45 | 34.26 | 56.61 | 60.91 |
| 16384 | 103.76 | 101.24 | 363.99 | 366.93 | 128.44 | 119.69 | 383.18 | 387.45 |

Table 5.9: Tests 1 and 2, $h = 5$.

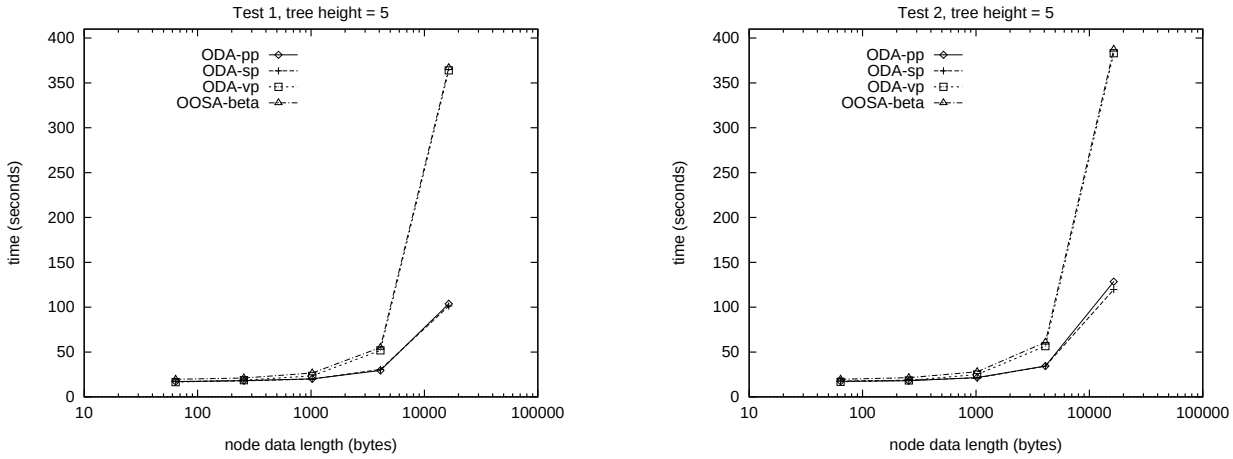



Figure 5.9: Tests 1 (left) and 2 (right), $h = 5$.

### 5.4.2 Database Traversal

We tested both a tree traversal that happens entirely at the server side, triggered by a single request from the client, and the case of a client-controlled traversal, in which the client actually visits every node of the tree.

**Test 3:** *Invoke the attribute accessor* `Node::descendant_count` *on the root node of a tree.*

Tables 5.10–5.12 and Figures 5.10–5.12 show the cold and warm times for this test; the warm times correspond to a second tree traversal. A traversal of the whole tree is our "cache warming" procedure: we measure warm times immediately after a tree traversal in all tests that follow.

Note that the response times for ODA-pp are actually ObjectStore numbers (plus about 2.2 milliseconds due to one remote method invocation). The cost of smart pointer-based persistence, though still small, appears clearly in the numbers for ODA-sp. The gap between these servers and the ones that use ObjectStore access hooks (ODA-vp and OOSA-$\beta$) is much larger now.

Because its page faulting scheme activates all objects in a page whenever one of them is touched, virtual persistence is expected to perform worse when database pages are shared by several objects and just a few objects per page are actually used. This is not the case here: since Test 3 does an exhaustive database traversal, all objects activated are eventually used. The bad performance of virtual persistence in

| | cold time ($\ell = 4096$) | | | | warm time ($\ell = 4096$) | | | |
|---|---|---|---|---|---|---|---|---|
| $h$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 1 | 0.06 | 0.08 | 0.14 | 0.26 | 0.02 | 0.02 | 0.13 | 0.22 |
| 2 | 0.18 | 0.22 | 0.54 | 0.93 | 0.04 | 0.06 | 0.42 | 0.81 |
| 3 | 0.68 | 0.81 | 2.04 | 2.94 | 0.10 | 0.14 | 1.52 | 2.44 |
| 4 | 2.88 | 3.57 | 8.25 | 10.93 | 0.39 | 0.53 | 6.07 | 8.90 |
| 5 | 11.01 | 13.62 | 31.70 | 39.51 | 1.46 | 2.14 | 23.56 | 32.13 |

Table 5.10: Test 3 — cold and warm times, $\ell = 4096$.

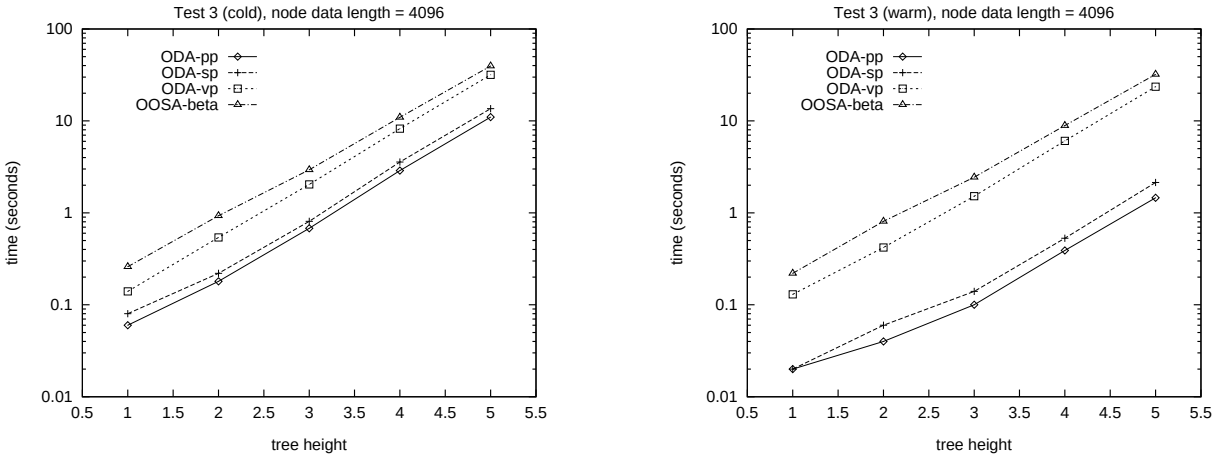



Figure 5.10: Test 3 — cold and warm times, $\ell = 4096$.

| | cold time ($h = 4$) | | | | warm time ($h = 4$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 0.13 | 0.67 | 0.50 | 0.37 | 0.04 | 0.17 | 0.35 | 0.29 |
| 256 | 0.34 | 0.82 | 0.79 | 0.94 | 0.06 | 0.21 | 0.56 | 0.71 |
| 1024 | 0.80 | 1.35 | 2.07 | 2.98 | 0.12 | 0.26 | 1.42 | 2.24 |
| 4096 | 2.88 | 3.57 | 8.25 | 10.93 | 0.39 | 0.53 | 6.07 | 8.90 |
| 16384 | 3.06 | 3.72 | 24.29 | 40.71 | 0.43 | 0.65 | 20.29 | 37.85 |

Table 5.11: Test 3 — cold and warm times, $h = 4$.

| | cold time ($h = 5$) | | | | warm time ($h = 5$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 0.48 | 2.61 | 1.84 | 1.15 | 0.12 | 0.64 | 1.52 | 0.66 |
| 256 | 1.06 | 3.29 | 3.23 | 3.34 | 0.20 | 0.75 | 2.37 | 2.45 |
| 1024 | 3.27 | 5.61 | 8.38 | 11.64 | 0.47 | 1.07 | 5.82 | 8.79 |
| 4096 | 11.01 | 13.62 | 31.70 | 39.51 | 1.46 | 2.14 | 23.56 | 32.13 |
| 16384 | 13.45 | 16.62 | 93.43 | 168.35 | 1.82 | 2.47 | 81.77 | 162.09 |

Table 5.12: Test 3 — cold and warm times, $h = 5$.

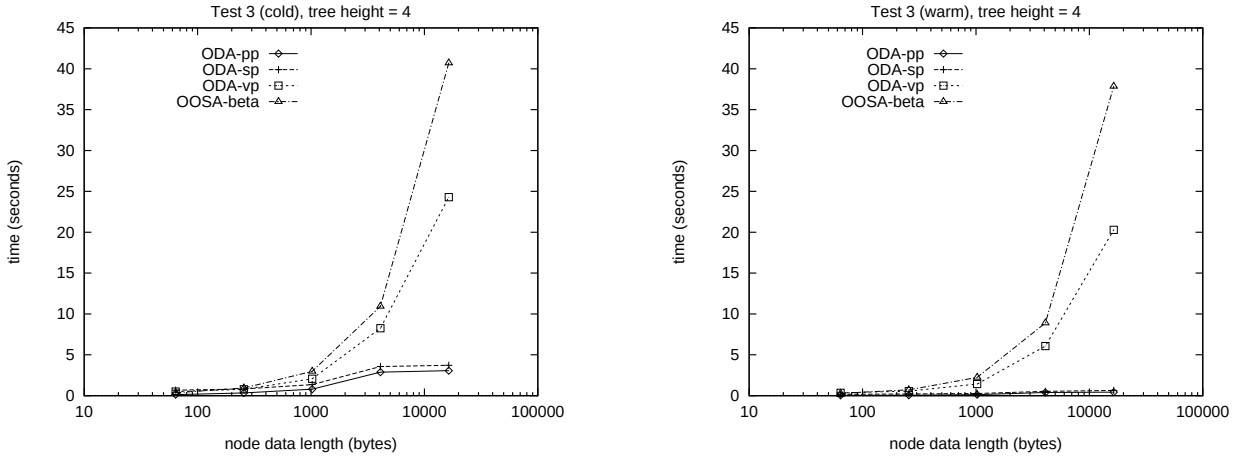



Figure 5.11: Test 3 — cold and warm times, $h = 4$.

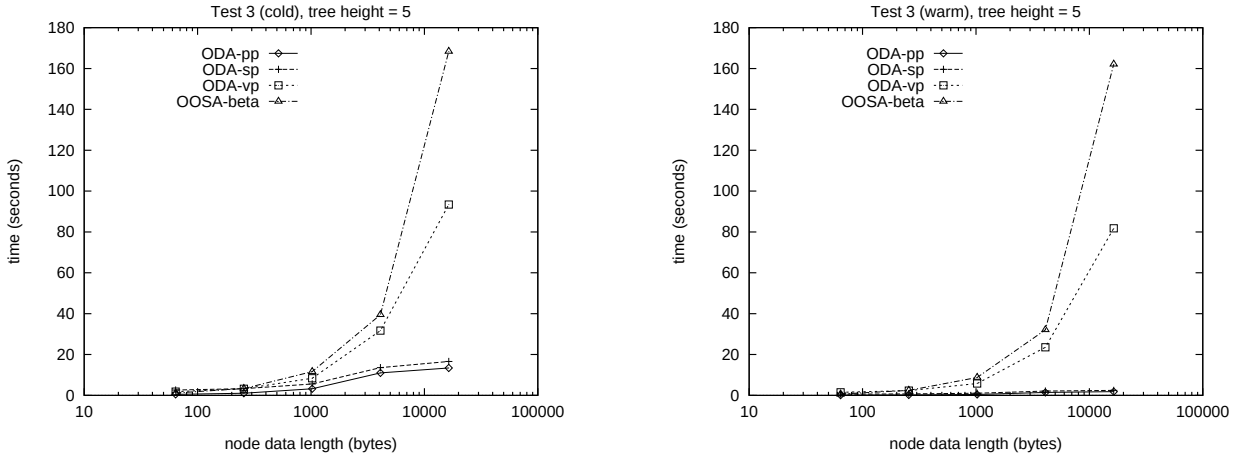



Figure 5.12: Test 3 — cold and warm times, $h = 5$.

Test 3 is due exclusively to the overhead of ObjectStore access hooks.

**Test 4:** *Obtain the number of nodes of a tree by recursively visiting all the nodes, through a sequence of calls to the attribute accessor* `Node::children`*. Perform the whole tree traversal within a single transaction.*

Since the tree traversal is now controlled by the client through remote method invocations, the overall response time now has two components: the database traversal time (given by Test 3) and the communication time (ORB overheads included). Tables 5.13–5.15 and Figures 5.13–5.15 show the cold and warm times for this test.

| | cold time ($\ell = 4096$) | | | | warm time ($\ell = 4096$) | | | |
|---|---|---|---|---|---|---|---|---|
| $h$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 1 | 0.09 | 0.09 | 0.16 | 0.32 | 0.07 | 0.05 | 0.13 | 0.27 |
| 2 | 0.36 | 0.32 | 0.67 | 1.24 | 0.15 | 0.16 | 0.53 | 1.05 |
| 3 | 1.27 | 1.29 | 2.62 | 4.09 | 0.60 | 0.58 | 2.03 | 3.35 |
| 4 | 5.15 | 5.26 | 10.66 | 15.84 | 2.30 | 2.30 | 8.05 | 12.64 |
| 5 | 20.99 | 21.03 | 40.93 | 58.73 | 9.61 | 9.30 | 32.07 | 48.81 |

Table 5.13: Test 4 — cold and warm times, $\ell = 4096$.

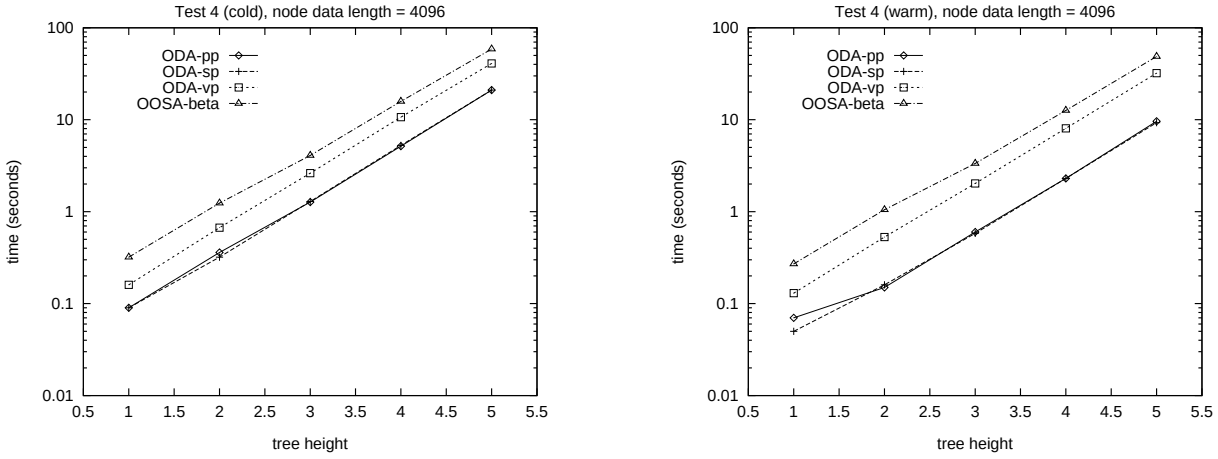



Figure 5.13: Test 4 — cold and warm times, $\ell = 4096$.

| | cold time ($h = 4$) | | | | warm time ($h = 4$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 2.38 | 2.41 | 2.56 | 3.26 | 1.87 | 1.88 | 2.35 | 2.99 |
| 256 | 2.51 | 2.61 | 2.90 | 3.97 | 1.89 | 1.86 | 2.55 | 3.57 |
| 1024 | 3.12 | 3.14 | 4.30 | 6.60 | 1.93 | 1.98 | 3.42 | 5.57 |
| 4096 | 5.15 | 5.26 | 10.66 | 15.84 | 2.30 | 2.30 | 8.05 | 12.64 |
| 16384 | 5.57 | 5.65 | 24.96 | 46.07 | 2.42 | 2.48 | 22.51 | 42.47 |

Table 5.14: Test 4 — cold and warm times, $h = 4$.

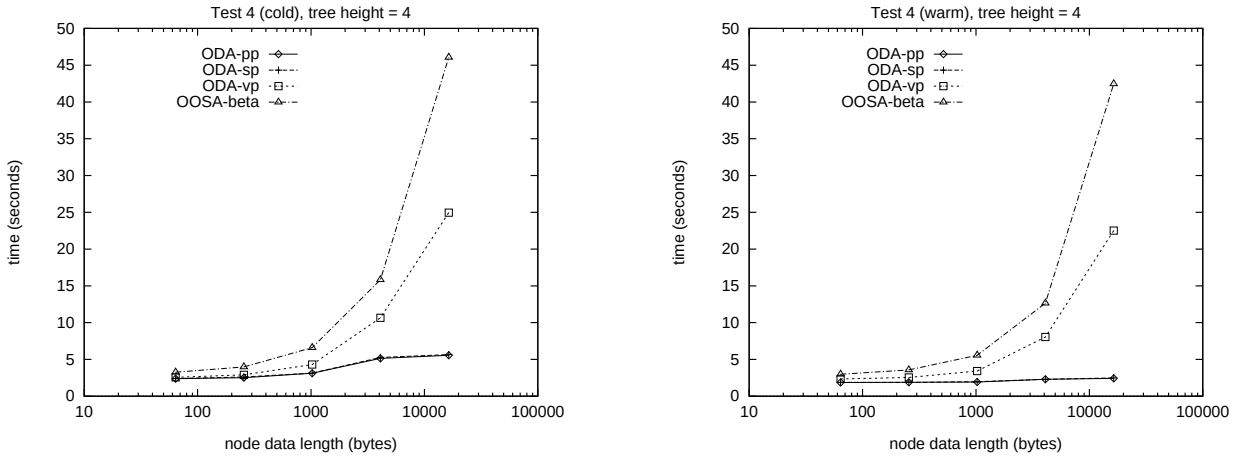



Figure 5.14: Test 4 — cold and warm times, $h = 4$.

| | cold time ($h = 5$) | | | | warm time ($h = 5$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 10.62 | 9.58 | 10.08 | 13.76 | 7.66 | 7.52 | 9.26 | 12.83 |
| 256 | 10.29 | 10.26 | 11.39 | 16.72 | 7.72 | 7.62 | 10.36 | 15.05 |
| 1024 | 12.57 | 12.86 | 17.18 | 27.06 | 7.94 | 7.96 | 13.97 | 22.67 |
| 4096 | 20.99 | 21.03 | 40.93 | 58.73 | 9.61 | 9.30 | 32.07 | 48.81 |
| 16384 | 23.15 | 23.73 | 101.43 | 198.32 | 22.79 | 24.70 | 108.04 | 203.86 |

Table 5.15: Test 4 — cold and warm times, $h = 5$.

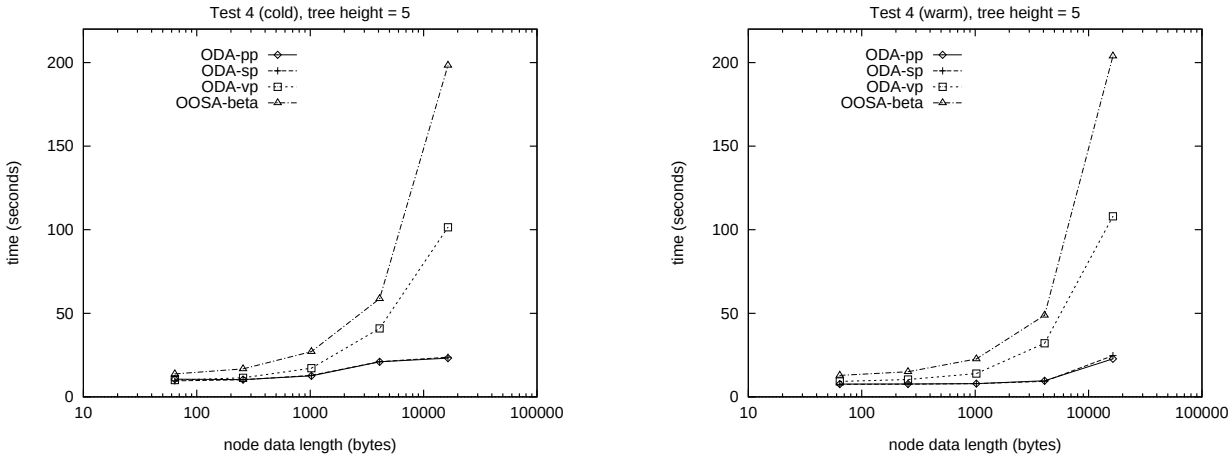



Figure 5.15: Test 4 — cold and warm times, $h = 5$.

**Test 5:** *Obtain the number of nodes of a tree by recursively visiting all the nodes, through a sequence of calls to the attribute accessor* `Node::children`*. Perform each call as an individual transaction.*

Now there is the additional overhead of one transaction per remote method invocation. Tables 5.16–5.18 and Figures 5.16–5.18 show the cold and warm times for this test. By comparing the results of Tests 3, 4, and 5, we see that the cost of transactions is either comparable to that of remote method invocations, or — in the case of servers ODA-vp and OOSA-$\beta$ — significantly larger.

| | cold time ($\ell = 4096$) | | | | warm time ($\ell = 4096$) | | | |
|---|---|---|---|---|---|---|---|---|
| $h$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 1 | 0.14 | 0.16 | 0.40 | 0.93 | 0.10 | 0.10 | 0.37 | 0.98 |
| 2 | 0.69 | 0.67 | 1.90 | 4.93 | 0.45 | 0.48 | 1.71 | 4.77 |
| 3 | 2.75 | 2.73 | 7.69 | 21.33 | 1.93 | 1.80 | 7.25 | 20.29 |
| 4 | 11.32 | 11.37 | 33.94 | 90.90 | 7.73 | 7.70 | 31.71 | 88.10 |
| 5 | 46.73 | 47.16 | 141.61 | 388.34 | 31.41 | 30.07 | 133.10 | 373.24 |

Table 5.16: Test 5 — cold and warm times, $\ell = 4096$.

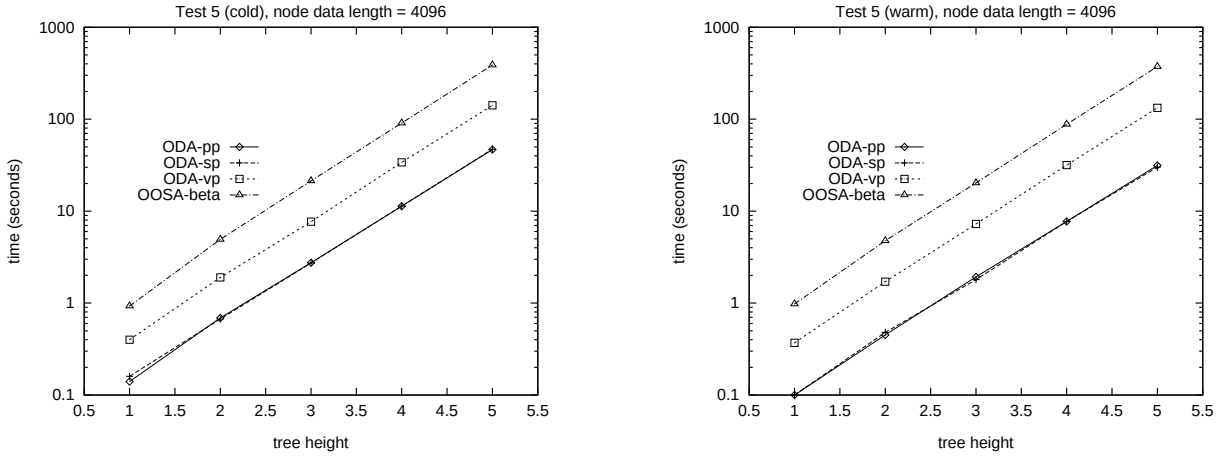



Figure 5.16: Test 5 — cold and warm times, $\ell = 4096$.

| | cold time ($h = 4$) | | | | warm time ($h = 4$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 7.27 | 7.28 | 21.92 | 50.73 | 7.60 | 6.70 | 22.00 | 51.26 |
| 256 | 7.61 | 7.75 | 19.23 | 60.34 | 7.39 | 6.81 | 19.36 | 59.77 |
| 1024 | 8.34 | 8.46 | 19.82 | 66.69 | 7.43 | 7.13 | 19.27 | 66.37 |
| 4096 | 11.32 | 11.37 | 33.94 | 90.90 | 7.73 | 7.70 | 31.71 | 88.10 |
| 16384 | 11.65 | 11.77 | 72.71 | 188.18 | 8.03 | 7.35 | 70.05 | 182.83 |

Table 5.17: Test 5 — cold and warm times, $h = 4$.

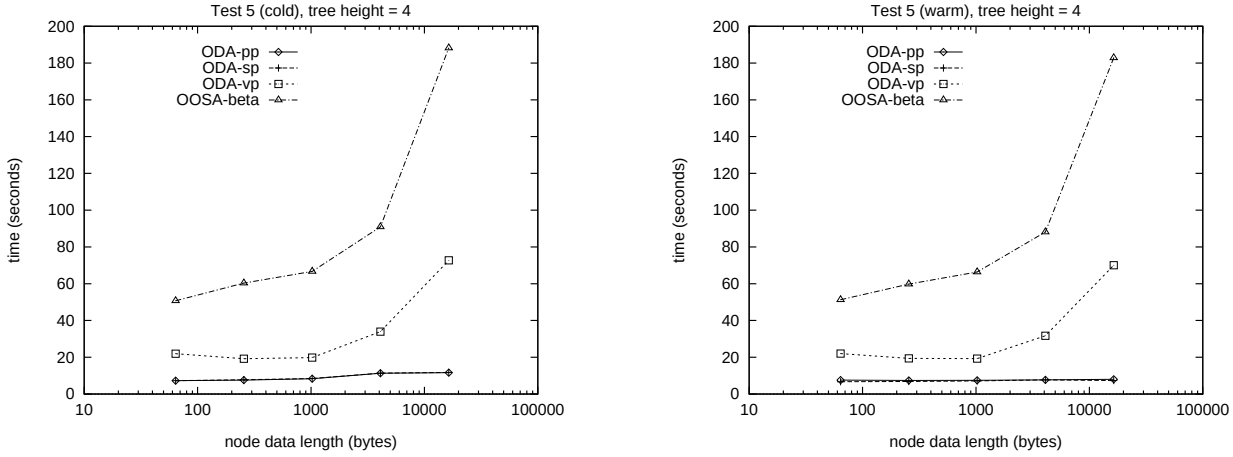



Figure 5.17: Test 5 — cold and warm times, $h = 4$.

| $\ell$ | cold time ($h = 5$) | | | | warm time ($h = 5$) | | | |
|---|---|---|---|---|---|---|---|---|
| | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 30.16 | 30.24 | 88.18 | 251.03 | 29.24 | 27.81 | 88.97 | 251.83 |
| 256 | 31.01 | 30.89 | 80.22 | 268.35 | 29.14 | 28.43 | 79.87 | 266.25 |
| 1024 | 35.52 | 35.03 | 84.41 | 309.46 | 30.50 | 28.43 | 80.90 | 294.05 |
| 4096 | 46.73 | 47.13 | 141.61 | 388.34 | 31.41 | 30.07 | 133.10 | 373.24 |
| 16384 | 50.49 | 50.8 | 300.36 | 866.85 | 32.31 | 31.04 | 291.55 | 843.89 |

Table 5.18: Test 5 — cold and warm times, $h = 5$.

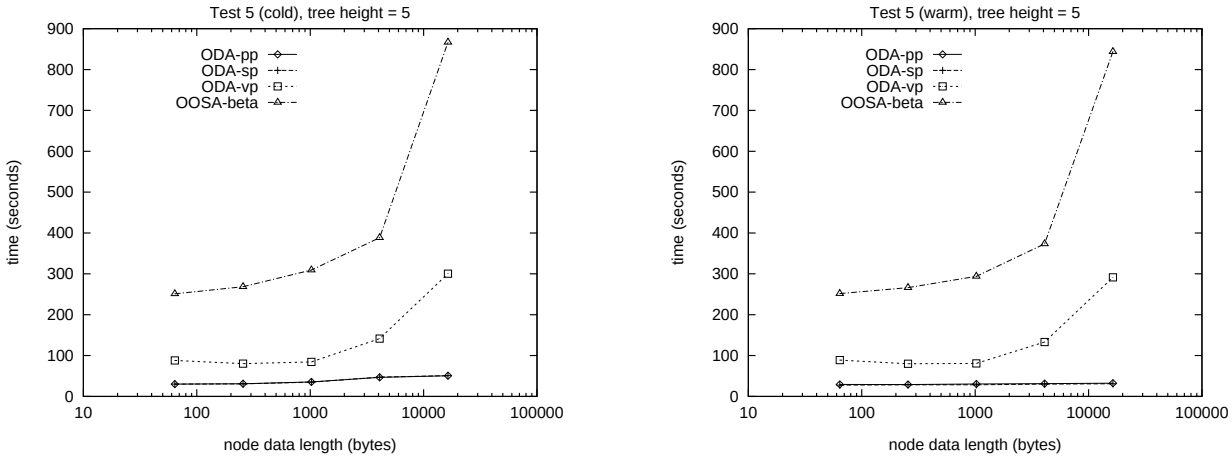



Figure 5.18: Test 5 — cold and warm times, $h = 5$.

### 5.4.3 Database Search

The tests reported so far stressed the performance differences among the servers. However, they are not representative of usual database access patterns. Rather than creating or traversing an entire database at once, most applications search a database for specific items to be retrieved or updated. The next two tests perform a client-controlled search, using the observation below.

**Observation 1:** If the node named `"Node(`$s$`)"` is a descendant of the node named `"Node(`$t$`)"`, then the string $t$ is a prefix of the string $s$.

**Test 6:** *Search a tree for a node with a given name, through a sequence of calls to the attribute accessors* `Node::children` *and* `Node::name`. *Rather than visiting every node, use Observation 1 to descend through the single tree branch that may contain a node with that name. Perform the whole search within a single transaction.*

**Test 7:** *Same as Test 6, but performing each call to the server as an individual transaction.*

We run these tests for a name that was not in the database, chosen to make the search process descend through the rightmost branch of the tree until it reaches a leaf[6].

Tables 5.19–5.21 and Figures 5.19–5.21 show the cold and warm times for Test 6. Tables 5.22–5.24 and Figures 5.22–5.24 show these times for Test 7. The results of Tests 6 and 7 are consistent with the previous tests, in that they confirm the relative position of each server in a performance line. The distances among servers, however, have decreased with respect to the preceding results. The major cause of the performance gap between the two pairs of servers is the cost of ObjectStore hooks, which hits only ODA-vp and OOSA-$\beta$. Since the amount of persistent memory accessed by Tests 6 and 7 is smaller in comparison with the preceding tests, the overhead of access hooks has a correspondingly smaller — but still quite significant

[6]With the convention we adopted for `Node` names, this is a name of the form `"Node(0,3,3,...,3)"`, where the sequence of `3`s has length greater than the height of the tree.

— effect. Note that ODA-vp performs better than OOSA-$\beta$, simply because it uses an inbound hook, whose cost is about half of the cost of the outbound hook used by OOSA-$\beta$.

| | cold time ($\ell = 4096$) | | | | warm time ($\ell = 4096$) | | | |
|---|---|---|---|---|---|---|---|---|
| $h$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 1 | 0.09 | 0.10 | 0.19 | 0.34 | 0.06 | 0.06 | 0.14 | 0.28 |
| 2 | 0.16 | 0.16 | 0.33 | 0.67 | 0.09 | 0.08 | 0.29 | 0.61 |
| 3 | 0.25 | 0.23 | 0.46 | 0.89 | 0.12 | 0.13 | 0.37 | 0.78 |
| 4 | 0.33 | 0.30 | 0.68 | 1.22 | 0.19 | 0.17 | 0.54 | 1.06 |
| 5 | 0.43 | 0.44 | 0.87 | 1.50 | 0.21 | 0.18 | 0.68 | 1.21 |

Table 5.19: Test 6 — cold and warm times, $\ell = 4096$.

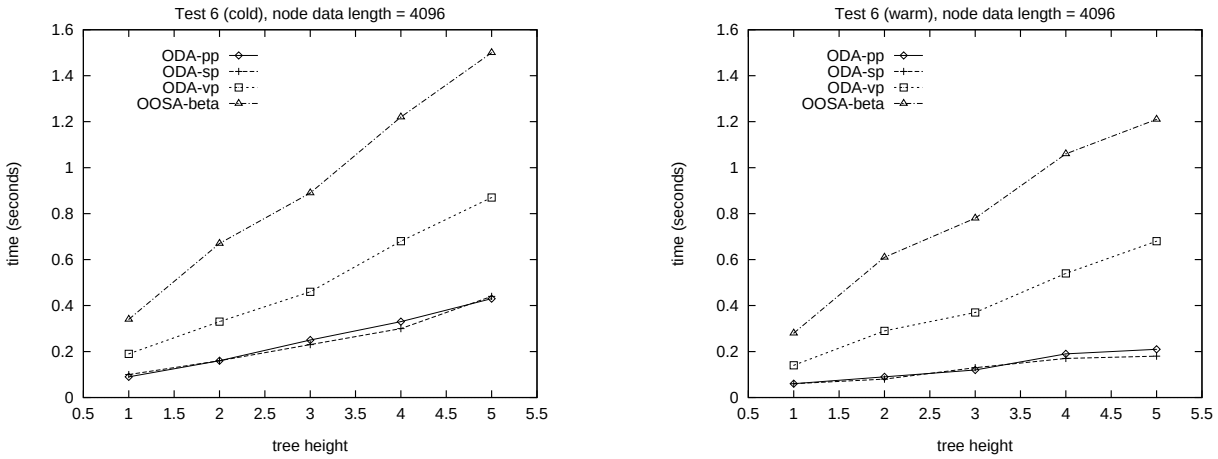



Figure 5.19: Test 6 — cold and warm times, $\ell = 4096$.

| | cold time ($h = 4$) | | | | warm time ($h = 4$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 0.18 | 0.20 | 0.28 | 0.40 | 0.13 | 0.13 | 0.23 | 0.39 |
| 256 | 0.23 | 0.21 | 0.29 | 0.46 | 0.13 | 0.14 | 0.22 | 0.40 |
| 1024 | 0.23 | 0.24 | 0.38 | 0.68 | 0.18 | 0.14 | 0.31 | 0.52 |
| 4096 | 0.33 | 0.30 | 0.68 | 1.22 | 0.19 | 0.17 | 0.54 | 1.06 |
| 16384 | 0.34 | 0.34 | 1.61 | 2.89 | 0.17 | 0.15 | 1.31 | 2.60 |

Table 5.20: Test 6 — cold and warm times, $h = 4$.

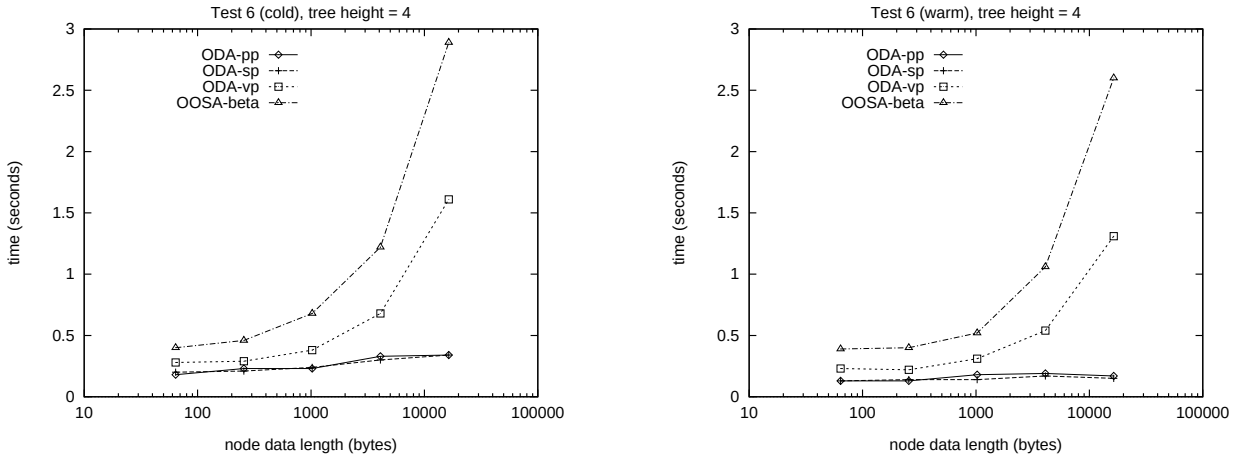



Figure 5.20: Test 6 — cold and warm times, $h = 4$.

| | cold time ($h = 5$) | | | | warm time ($h = 5$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 0.24 | 0.26 | 0.37 | 0.59 | 0.17 | 0.15 | 0.28 | 0.43 |
| 256 | 0.27 | 0.25 | 0.36 | 0.80 | 0.17 | 0.16 | 0.36 | 0.51 |
| 1024 | 0.31 | 0.32 | 0.50 | 0.86 | 0.16 | 0.18 | 0.36 | 0.69 |
| 4096 | 0.43 | 0.44 | 0.87 | 1.50 | 0.21 | 0.18 | 0.68 | 1.21 |
| 16384 | 0.46 | 0.45 | 1.98 | 3.70 | 0.32 | 0.19 | 1.79 | 3.43 |

Table 5.21: Test 6 — cold and warm times, $h = 5$.

| | cold time ($\ell = 4096$) | | | | warm time ($\ell = 4096$) | | | |
|---|---|---|---|---|---|---|---|---|
| $h$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 1 | 0.20 | 0.20 | 0.45 | 1.24 | 0.13 | 0.14 | 0.40 | 1.14 |
| 2 | 0.36 | 0.33 | 0.78 | 2.66 | 0.26 | 0.25 | 0.72 | 2.51 |
| 3 | 0.48 | 0.49 | 1.17 | 3.96 | 0.32 | 0.34 | 1.04 | 3.84 |
| 4 | 0.68 | 0.65 | 1.68 | 5.59 | 0.46 | 0.44 | 1.49 | 5.44 |
| 5 | 0.84 | 0.86 | 2.20 | 7.48 | 0.52 | 0.53 | 1.98 | 6.78 |

Table 5.22: Test 7 — cold and warm times, $\ell = 4096$.

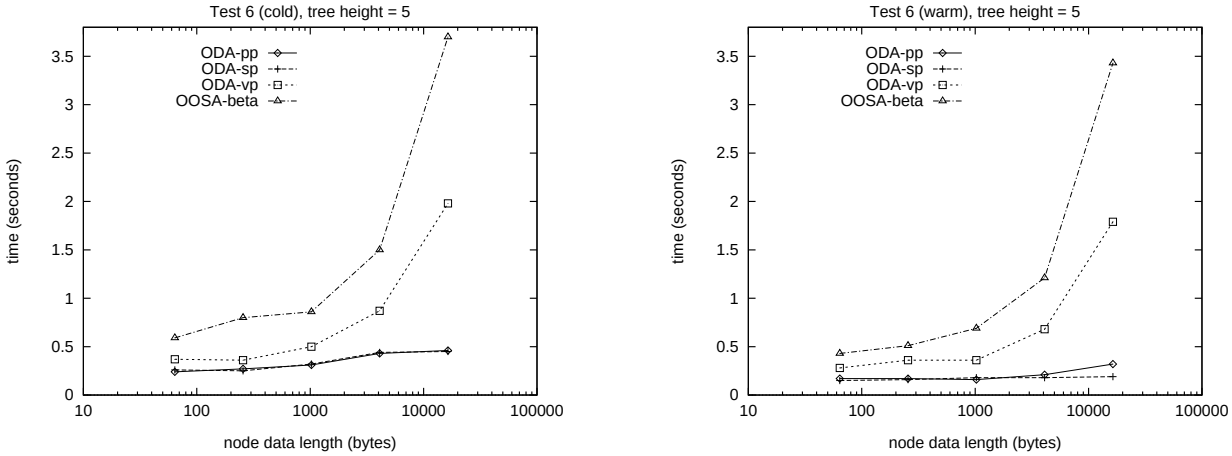



Figure 5.21: Test 6 — cold and warm times, $h = 5$.

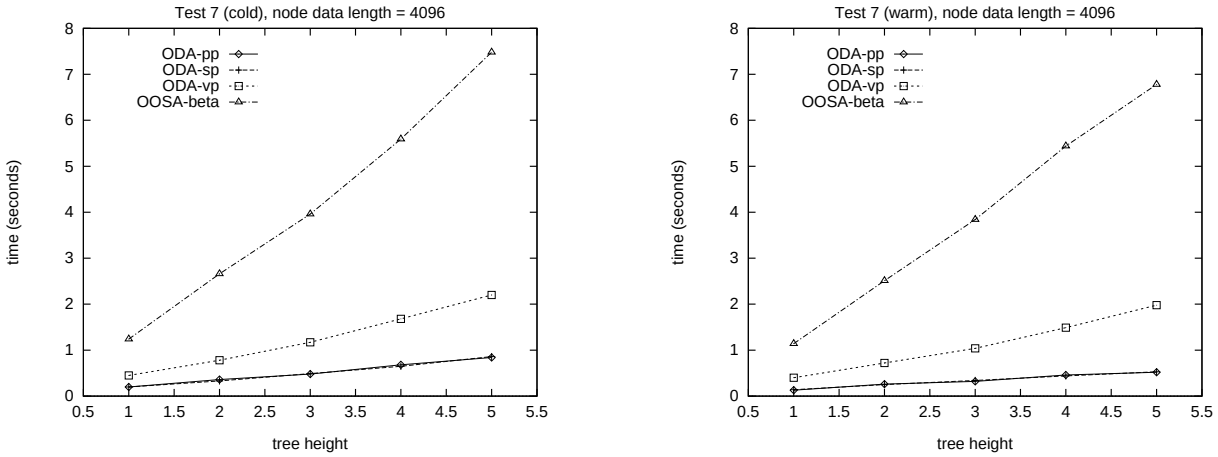
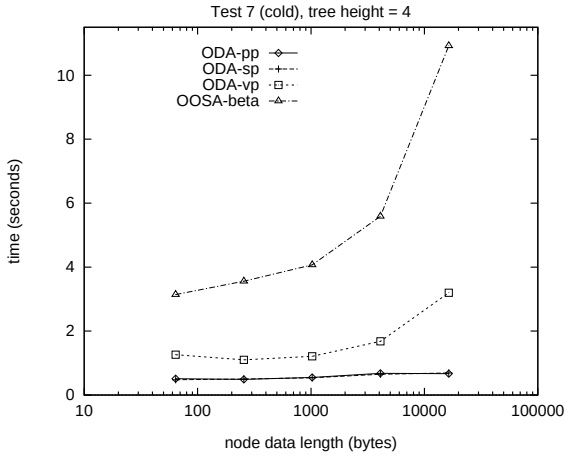



Figure 5.22: Test 7 — cold and warm times, $\ell = 4096$.

| | cold time ($h = 4$) | | | | warm time ($h = 4$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 0.51 | 0.48 | 1.26 | 3.14 | 0.42 | 0.42 | 1.32 | 3.05 |
| 256 | 0.49 | 0.50 | 1.10 | 3.56 | 0.42 | 0.44 | 1.07 | 3.54 |
| 1024 | 0.55 | 0.54 | 1.21 | 4.07 | 0.41 | 0.41 | 1.05 | 3.96 |
| 4096 | 0.68 | 0.65 | 1.68 | 5.59 | 0.46 | 0.44 | 1.49 | 5.44 |
| 16384 | 0.67 | 0.69 | 3.20 | 10.92 | 0.44 | 0.45 | 3.03 | 10.65 |

Table 5.23: Test 7 — cold and warm times, $h = 4$.



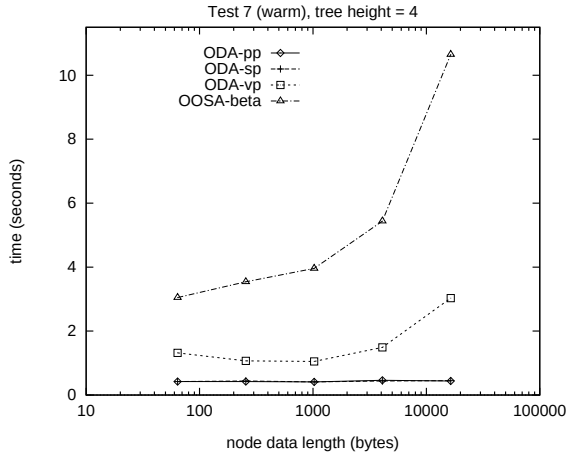
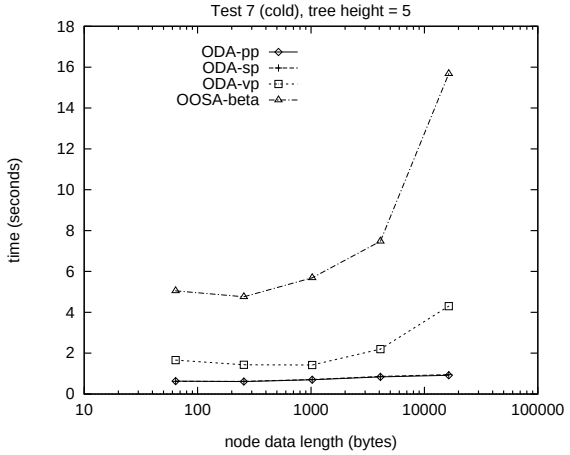




Figure 5.23: Test 7 — cold and warm times, $h = 4$.

| | cold time ($h = 5$) | | | | warm time ($h = 5$) | | | |
|---|---|---|---|---|---|---|---|---|
| $\ell$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ | ODA-pp | ODA-sp | ODA-vp | OOSA-$\beta$ |
| 64 | 0.63 | 0.62 | 1.66 | 5.05 | 0.57 | 0.53 | 1.60 | 4.51 |
| 256 | 0.61 | 0.62 | 1.43 | 4.76 | 0.55 | 0.60 | 1.45 | 4.67 |
| 1024 | 0.70 | 0.71 | 1.42 | 5.69 | 0.55 | 0.54 | 1.38 | 5.48 |
| 4096 | 0.84 | 0.86 | 2.20 | 7.48 | 0.52 | 0.53 | 1.98 | 6.78 |
| 16384 | 0.92 | 0.95 | 4.30 | 15.68 | 0.58 | 0.56 | 4.18 | 16.99 |

Table 5.24: Test 7 — cold and warm times, $h = 5$.



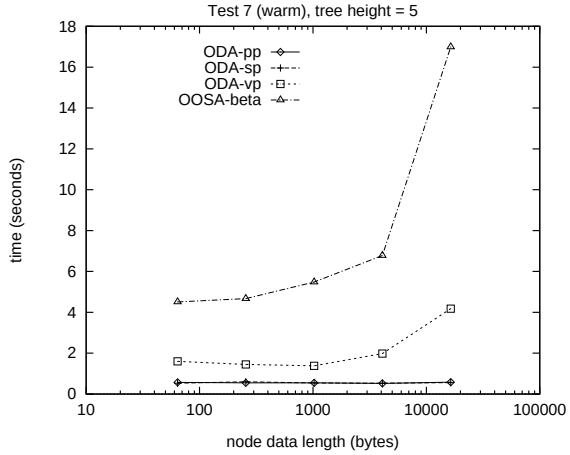




Figure 5.24: Test 7 — cold and warm times, $h = 5$.

### 5.4.4 Discussion

We now summarize and discuss our results.

**ORB/ODBMS integration is a viable and balanced approach.**

The gap between object access times in the ORB and in the ODBMS environment is sometimes used as an argument against ORB/ODBMS integration. In this view, the ORB remote method invocation mechanism would be an access procedure unacceptably slow, as compared to the ODBMS data shipping approach. The ORB would introduce a tight bottleneck in a system that would have much higher performance otherwise. Our results show that this is not the case.

**Database transactions cost more than remote method invocations.** Tests 6 and 7 use the same sequence of remote method invocations to perform a client-controlled search. Test 6 performs the whole sequence of operations within a single transaction; Test 7 executes each operation as an individual transaction. The pseudopersistence results of both tests are plotted together in Figure 5.25, for $\ell = 4096$. Note that the transaction costs (the differences between the $y$ coordinates of the upper and lower points plotted) exceed the times for Test 6, which account for the remote method invocations *and* for the actual execution of the operations invoked.

Whenever possible, IDL operations should be designed to correspond to database transactions. If each operation performs the maximum amount of work that can be

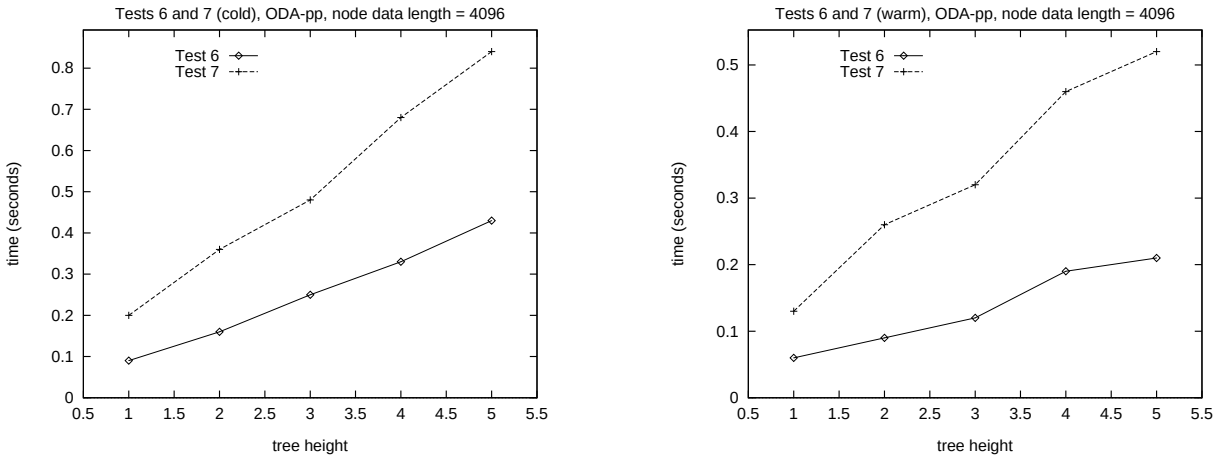



Figure 5.25: Tests 6 and 7, cold and warm pseudopersistence results, $\ell = 4096$.

done in a single transaction, then remote method invocation costs are dominated by transaction costs. In some situations this goal may be unrealistic, and the operations may end up doing much less than could be done in a transaction. If this is the case, CORBA clients will achieve better performance by grouping multiple operations within a single transaction.

**Pseudopersistence has good performance.**

Pseudopersistence consistently outperformed the other approaches tested. It delivers ODBMS performance in the case of a database-intensive task executed as a single operation (Test 3). For operations that are not database intensive, but need to be executed as individual transactions, pseudopersistence achieves at least half of the ODBMS performance (Tests 5 and 7).

**Object caching is effective.** For small objects, caching decreased the times of Test 1 by a factor of 0.5–0.6. For larger objects (`Nodes` with data length 16384), the corresponding factor was approximately 0.9. Since the time saved by caching tie objects is independent of the size of the corresponding implementation objects, caching yields greater benefits for finer-grained objects.

**Smart pointer-based persistence is a promising approach.**

Even though the *actual* performance of smart pointer-based persistence on ObjectStore is unsatisfactory, its *hypothetical* performance, which would be achieved with adequate ODBMS support, is good. ODBMS products are expected to provide such support in the near future. When this happens, smart pointer-based persistence should perform almost as well as pseudopersistence.

**Virtual persistence needs better ODBMS support.**

Due to the prohibitive overhead of ObjectStore access hooks, virtual persistence showed poor performance. With better ODBMS support, access hook overhead could — and, we expect, eventually will — be considerably smaller. Until then, one cannot seriously consider using virtual persistence for practical purposes.

Besides rendering virtual persistence useless, access hook overhead made some of our results less interesting. Our virtual persistence numbers reflect an ODBMS problem, not the essential performance characteristics of this approach. Two factors affect the intrinsic performance of this virtual persistence:

1. The cost of activating objects that will not be actually used. Because virtual persistence is a page faulting scheme, it may give you more than you need, and will charge you for it. This cost is expected when database pages are shared by several objects, and the database client uses only a few objects per page.

2. The essential, and unavoidable, cost of access hooks. The ODBMS must keep track of which hook should be called for each object. It maintains data structures that associate a hook with objects of a given class.

To these factors, the ObjectStore implementation of access hooks added a superfluous, and dominant, one:

3. The cost of flushing out and reloading the client cache at every transaction.

Unfortunately, the intrinsic performance of virtual persistence cannot be inferred from our results. We considered estimating it by appraising the values of cost 3 and subtracting them from the virtual persistence numbers; disk accesses could be used to appraise these costs. The problem, however, is that cost 3 is the dominant one. Even small errors in its appraised values would have a great impact on the estimated performance numbers. For this reason, the estimated intrinsic performance would not be reliable.

**A final experiment.** If we could evaluate costs 1 and 2, we would be able to estimate the intrinsic performance of virtual persistence by adding these costs to the smart pointer-based persistence numbers. In the case of Tests 3 and 4, which perform an exhaustive tree traversal within a single transaction, cost 1 is zero, because all objects in the database are accessed by the transaction. Cost 2, however, is unknown. One could think about adding a dummy access hook, which performs no work, to server ODA-sp. By doing this, we would add cost 2, as we want, but would also add cost 3, which we do not want! So a dummy hook would bring ODA-sp numbers to the level of ODA-vp numbers. As a final test, we did such an experiment.

Table 5.25 presents the results of adding a dummy hook to server ODA-sp. To mimic virtual persistence, a dummy "`Node` representative" was stored with each `Node_impl` object; these dummy representatives have the same size as the object representatives used in virtual persistence. An inbound hook is associated with the dummy representatives, exactly as with object representatives in virtual persistence. In this experiment, however, the inbound hook performs no work, and the dummy representatives are not used by the CORBA server.

| $h$ | Test 3 | | Test 4 | | Test 5 | |
|---|---|---|---|---|---|---|
| | cold | warm | cold | warm | cold | warm |
| 1 | 0.14 | 0.10 | 0.18 | 0.13 | 0.31 | 0.27 |
| 2 | 0.61 | 0.40 | 0.67 | 0.54 | 1.54 | 1.28 |
| 3 | 2.12 | 1.40 | 2.53 | 1.91 | 6.19 | 5.38 |
| 4 | 8.20 | 5.48 | 10.04 | 7.38 | 25.08 | 21.94 |
| 5 | 35.04 | 23.98 | 43.46 | 31.79 | 105.63 | 88.70 |

Table 5.25: Tests 3, 4, and 5 — ODA-sp with dummy inbound hook, $\ell = 4096$.

Figures 5.26–5.28 show the results of this experiment along with the corresponding ODA-vp numbers. As expected, ODA-sp with a dummy hook performs the same as ODA-vp in Tests 3 and 4 (Figures 5.26 and 5.27). Note that does not happen in Test 5 (Figure 5.28), because cost 1 is not zero in this case. Test 5 performs a client-controlled database traversal, like Test 4. In Test 5, however, each operation is performed as an individual transaction, which accesses just a fraction of the objects in the database. Since the inbound hook of server ODA-vp activates objects that will not be used by the current transaction, this server performs worse than ODA-sp with a dummy inbound hook — the dummy hook activates no objects at all.

This experiment, which should remove any doubts one might have about why server ODA-vp performs so badly, exposed once more the penalty for using an ObjectStore access hook, even one that does no work, and served as a final sanity check on our numbers. To the end of assessing the intrinsic performance of virtual persistence, however, it did not help us much. It is clear that virtual persistence can perform better. How much better? Could it compete with the other approaches? The answers will remain unknown until adequate ODBMS support is available.

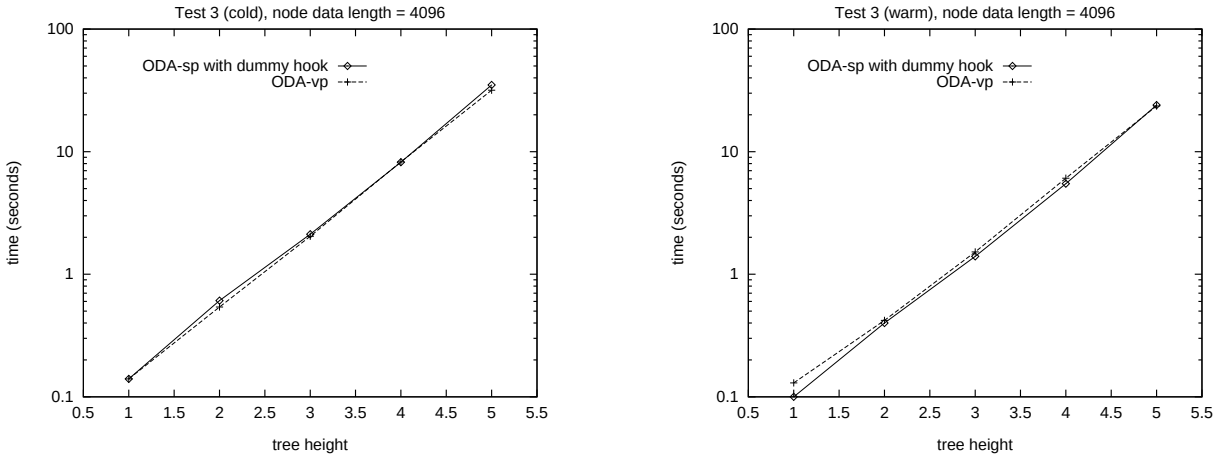



Figure 5.26: Test 3, ODA-sp with dummy inbound hook and ODA-vp, $\ell = 4096$.

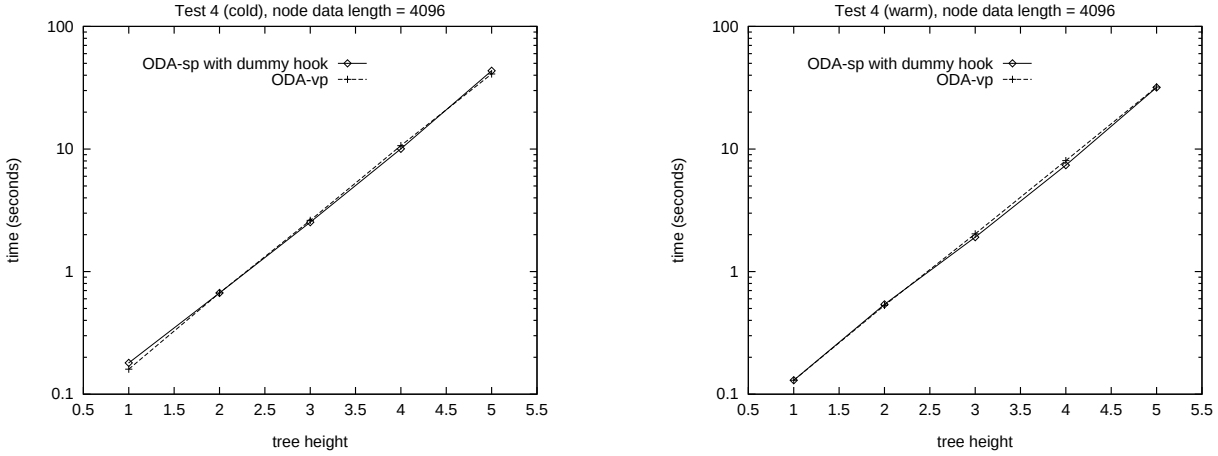



Figure 5.27: Test 4, ODA-sp with dummy inbound hook and ODA-vp, $\ell = 4096$.

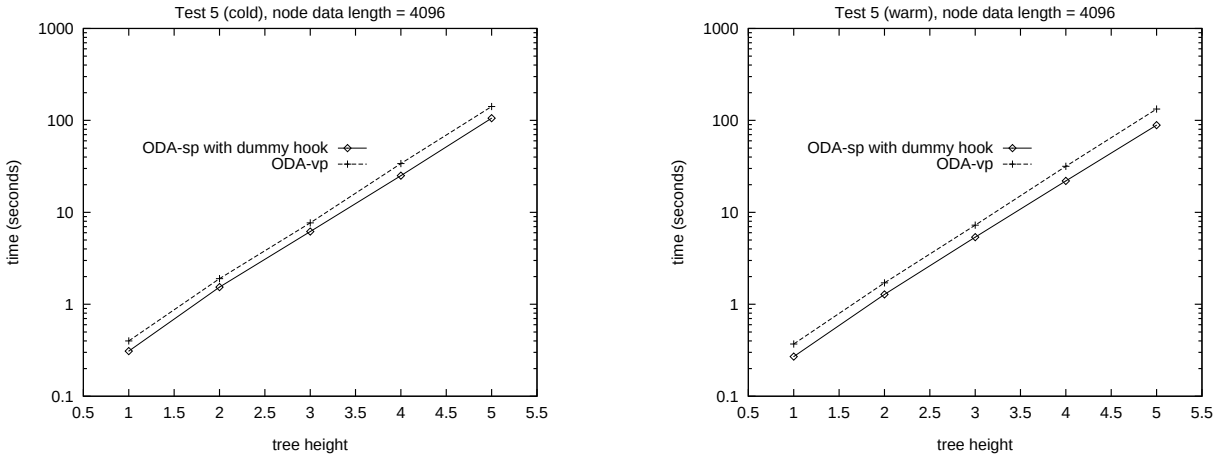



Figure 5.28: Test 5, ODA-sp with dummy inbound hook and ODA-vp, $\ell = 4096$.

# Chapter 6

# Conclusion

This dissertation presented a detailed discussion of the design and implementation issues involved in integrating Object Request Brokers and Object Database Management Systems. Three approaches to ODBMS-based persistence of CORBA objects were introduced: pseudopersistence, smart pointer-based persistence and virtual persistence.

These approaches share a common feature: their use of a delegation relationship between IDL-generated skeleton (or "tie") classes and user-provided implementation classes, with the purpose of allowing implementation objects to be stored in a database without the corresponding tie objects. Tie objects are not stored; they are automatically instantiated by an Object Database Adapter whenever they are needed and deleted when not needed. To make possible the instantiation of tie objects, the Object Database Adapter embeds a stringfied ODBMS reference to the corresponding implementation object into the `id` field of a CORBA reference to a persistent object.

The three approaches differ in what may cause the instantiation of a tie object. Pseudopersistence uses the CORBA object activation mechanism, which works only for object references received from other processes or stored in stringfied form. Smart pointer-based persistence extends pseudopersistence with a smart pointer scheme; virtual persistence extends it by employing virtual memory techniques.

Besides benefits with respect to database space consumption and performance, the use of delegation ("tie" objects) to keep only implementation objects — and not their corresponding CORBA objects — in the database has an important and desirable consequence: no ORB-specific information is stored persistently. Other

solutions to the problem of ODBMS-based persistence of CORBA objects are possible, but they store ORB-specific information in the database. A change of ORB, or even modifications to the ORB, would then affect the database schema. Surprisingly enough, it appears that this very simple idea is a contribution of this dissertation. Other contributions are:

- The notion of an Object Database Adapter as a library that uses and extends the services provided by the Basic Object Adapter, rather than as replacement to the BOA. This is not an original contribution. Related (and unpublished) work was simultaneously carried out at Iona Technologies [19], and probably at other software companies.

- An in-depth discussion of the aforementioned approaches to ODBMS-based persistence of CORBA objects. To the best of our knowledge, these issues were not addressed elsewhere. As described here, pseudopersistence relies exclusively on very basic ODBMS features; an Object Database Adapter based upon this approach can be easily ported across ODBMSs. Moreover, our realization of virtual persistence, through minimum-sized persistent representatives that are swizzled into CORBA objects at page fault time, appears to be an original idea.

- A performance evaluation of ORB/ODBMS integration approaches. Our results show pseudopersistence as the best performer, followed closely by smart pointer-based persistence (under the assumption that the ODBMS provides adequate support to this approach). Virtual persistence, which applies only to the case of a virtual memory-based ODBMS, currently lags well behind the other approaches; it will remain in this position until better ODBMS support is available.

## 6.1 Summary of Results

1. CORBA access to databases is viable and compatible with the performance of transactional database applications. ORB/ODBMS integration is the best way of providing database access to CORBA clients. Besides being directly applicable to pure object-oriented DBMSs, object-relational mediators make

it also applicable to relational systems. Moreover, ORB/ODBMS integration allows the construction of ORB-connected multidatabases, and is especially suitable for web browser access, through Java applets.

2. Among the approaches to persistence of CORBA objects, pseudopersistence is unquestionably the best option today. It works with any ODBMS, stores no ORB-specific information in a database, and has good performance.

3. ODMG-compliant ODBMSs are expected to make smart pointer-based persistence practical in the near future, and this approach will then be challenging pseudopersistence. It has all the advantages of pseudopersistence, plus transparent storability of CORBA object references. Although not essential, this feature is desirable in the context of ORB-connected multidatabases.

As for which of these two approaches will be the most used, this will depend on the degree of interconnectivity between the heterogeneous components of the persistent object systems that we expect to become common. For CORBA objects implemented by a single server, pseudopersistence is sufficient. Smart pointer-based persistence provides additional convenience to express relationships between CORBA objects implemented by different servers.

## 6.2 Future Work

We intend to apply the techniques discussed in this dissertation to ODMG-compliant ODBMSs and to the construction of ORB-connected multidatabases. Such a multidatabase could encompass legacy data stored in relational systems, and would be accessible through a web browser running Java applets.

A number of issues barely touched here will arise in the context of an ORB-connected multidatabase, including distributed transactions and queries. Implementations of the Object Transaction Service are starting to appear, but Object Query Service implementations are yet to be built. Efficiently implementing distributed queries in a CORBA environment will be a challenging and important research subject for some time.

# Glossary of Acronyms

| | |
|---|---|
| **BOA** | Basic Object Adapter |
| **CAD** | Computer-Aided Design |
| **CAM** | Computer-Aided Manufaturing |
| **CASE** | Computer-Aided Software Engineering |
| **CLOS** | Common Lisp Object System |
| **CORBA** | Common Object Request Broker Architecture |
| **DBMS** | Database Management System |
| **DII** | Dynamic Invocation Interface |
| **DTP** | Distributed Transaction Processing |
| **FIFO** | First In, First Out |
| **IDL** | Interface Definition Language |
| **IIOP** | Internet Inter-ORB Protocol |
| **IP** | Internet Protocol |
| **LAN** | Local Area Network |
| **LANL** | Los Alamos National Laboratory |
| **ODA** | Object Database Adapter |
| **ODBMS** | Object Database Management System |
| **ODMG** | Object Database Management Group |
| **OID** | Object Identifier |
| **OLE** | Object Linking and Embedding |
| **OMG** | Object Management Group |
| **OODBMS** | Object-Oriented Database Management System |

| | |
|---|---|
| **OOSA** | Orbix+ObjectStore Adapter |
| **ORB** | Object Request Broker |
| **OTS** | Object Transaction Service |
| **PDS** | Persistent Data Service |
| **PO** | Persistent Object |
| **POM** | Persistent Object Manager |
| **POS** | Persistent Object Service |
| **POSIX** | Portable Operating System Interface |
| **RPC** | Remote Procedure Call |
| **SQL** | The ANSI Standard Query Language |
| **TCP** | Transmission Control Protocol |
| **TP** | Transaction Processing |
| **XA** | The X/Open DTP resource manager interface |

# Bibliography


[1] Agrawal, R., and N. H. Gehani, "ODE (Object Database and Environment): The Language and the Data Model", *Proceedings of the 1989 ACM SIGMOD Conference on Management of Data*, Portland, OR, June 1989. Also in [54].

[2] AT&T GIS, DEC, Expersoft, Groupe Bull, HP, IBM, ICL, Novell, Siemens, SunSoft, Tandem, and Tivoli Systems, *CORBASecurity*, OMG Document 95-12-1, Object Management Group, Inc., Framingham, MA, 1995.

[3] Birrel, A., G. Nelson, S. Owicki, and E. Wobber, "Network Objects", *Proceedings of the 14th Symposium on Operating Systems Principles*, pp. 217–230, December 1993. An extended version of this paper appeared as Research Report 115, System Research Center, Digital Equipment Corp., Palo Alto, CA, 1994.

[4] Butterworth, P., A. Otis, and J. Stein, "The GemStone Object Database Management System", *Communications of the ACM*, Vol. 34, No. 10, pp. 64–77, October 1991.

[5] Carey, M. J., D. J. DeWitt, and J. F. Naughton, "The OO7 Object-Oriented Database Benchmark", *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, Washihgton, DC, May 1993.

[6] Cattell, R. G. G., *Object Data Management*, Addison-Wesley, Reading, MA, 1994.

[7] Cattell, R. G. G. (ed.), *The Object Database Standard: ODMG-93, Release 1.2*, Morgan Kaufmann, San Francisco, CA, 1996.

[8] Deux, O. *et al*, "The $O_2$ System", *Communications of the ACM*, Vol. 34, No. 10, pp. 34–48, October 1991.

[9] Deux, O. *et al*, "The Story of $O_2$", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 1, pp. 91-108, March 1990. Also in [54].

[10] Ellis, M., and B. Stroustroup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990.

[11] Gosling, J., and H. McGilton, *The Java Language Environment — A White Paper*, Sun Microsystems, Inc., Mountain View, CA, 1995.

[12] Gray, J., and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, CA, 1993.

[13] Hughes Technologies, *Mini SQL — A Lightweight Database Engine, Release 1.0.11*, Hughes Technologies Pty Ltd., Gold Coast, Australia, 1996.

[14] IBM Corp., *SOMobjects Developer Toolkit Programmers Reference Manual, Version 2.0*, IBM Corporation, White Plains, NY, 1993.

[15] IBM Corp., *SOMobjects Developer Toolkit Users Guide, Version 2.0*, IBM Corporation, White Plains, NY, 1993.

[16] IBM, Itasca, Objectivity, Ontos, O2, Servio, SunSoft, and Taligent, *Object Query Service Specification (Joint Submission to the OMG)*, OMG TC Document 95-11-1, Object Management Group, Inc., Framingham, MA, 1995.

[17] Iona Technologies, *Orbix Advanced Programmer's Guide*, Iona Technologies Ltd., Dublin, Ireland, 1995.

[18] Iona Technologies, *Orbix Programmer's Guide*, Iona Technologies Ltd., Dublin, Ireland, 1995.

[19] Iona Technologies, *White Paper — Orbix+ObjectStore Adapter*, Iona Technologies Ltd., Dublin, Ireland, 1995.

[20] Iona Technologies, *Orbix+ObjectStore Adapter — Beta Release Documentation*, Iona Technologies Ltd., Dublin, Ireland, 1995.

[21] Khoshafian, S., and G. P. Copeland, "Object Identity", *ACM Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Portland, OR, September 1986. Also in [61].

[22] Kim, W., and F. H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications*, ACM Press, New York, NY, 1989.

[23] Kim, W., N. Ballou, H. Chou, and J. F. Garza, "Features of the ORION Object-Oriented Database System", *in* [22].

[24] Keller, A. M., "Penguin: Objects for Programs, Relations for Persistence", submitted for publication, April 1994.

[25] Keller, A. M., and C. Hamon, "A C++ Binding for Penguin: a System for Data Sharing Among Heterogeneous Object Models", Foundations on Data Organization (FODO) 93, Chicago, October 1993.

[26] Keller, A. M., R. Jensen, and S. Agarwal, "Persistence Software: Bridging Object-Oriented Programming and Relational Databases", *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, Washihgton, DC, May 1993.

[27] Kim, W. (ed.), *Modern Database Systems — The Object Model, Interoperability and Beyond*, ACM Press, New York, NY, 1995.

[28] Kim, W., "Object-Oriented Database Systems: Promises, Reality, and Future", *in* [27].

[29] Lamb, C., G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System", *Communications of the ACM*, Vol. 34, No. 10, pp. 50–63, October 1991. Also in [54].

[30] Lewis, G., *CORBA Application Portability*, OMG TC Document 95-5-17, Object Management Group, Inc., Framingham MA, 1995.

[31] Malhotra, A., ODBMS presentation at the Los Alamos National Laboratory, July 1995.

[32] Manola, F., *An Evaluation of Object-Oriented DBMS Developments, 1994 Edition*, Technical Report TR-0263-08-94-165, GTE Laboratories Inc., Waltham, MA, 1994.

[33] Moss, J. E. B., "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Transactions on Software Engineering*, Vol. 18, No. 8, pp.657–673, August 1992.

[34] $O_2$ Technology, *The $O_2$ ODMG Database System — A Technical Overview*, $O_2$ Technology, Palo Alto, CA, 1995.

[35] Object Design, *ObjectStore/DBconnect*, Product Brief, Object Design, Inc., Burlington, MA, 1995.

[36] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91-12-1, Revision 1.1, Object Management Group, Inc., Framingham, MA, 1991.

[37] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, Object Management Group, Inc., Framingham, MA, 1995.

[38] Object Management Group, *CORBAfacilities: Common Object Facilities Specification*, Object Management Group, Inc., Framingham, MA, 1995.

[39] Object Management Group, *CORBAservices: Common Object Services Specification*, Object Management Group, Inc., Framingham, MA, 1995.

[40] Object Management Group, *Object Management Architecture Guide*, OMG TC Document 90-9-1, Revision 1.0, Object Management Group, Inc., Framingham MA, 1990.

[41] Object Management Group, *ORB Portability Enhancement RFP*, OMG TC Document 95-6-26, Object Management Group, Inc., Framingham MA, 1995.

[42] Objectivity, *Objectivity Database System Overview*, Objectivity, Inc., Menlo Park, CA, 1990.

[43] Ontos, *Ontos Reference Manual*, Ontos, Inc., Burlington, MA, 1993.

[44] Orfali, R., D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., New York, NY, 1996.

[45] PostModern Computing Technologies, *ORBeline 2.0 Reference Manual*, PostModern Computing Technologies, Inc., Mountain View, CA, 1995.

[46] PostModern Computing Technologies, *ORBeline 2.0 User's Guide*, PostModern Computing Technologies, Inc., Mountain View, CA, 1995.

[47] Richardson, J. E., M. J. Carey, and D. H. Schuh, "The Design of the E Programming Language", *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 3, July 1993. Also in [54].

[48] Richardson, J. E., and M. J. Carey, "Persistence in the E Language: Issues and Implementation", *Software — Practice and Experience*, Vol. 19, No. 12, pp. 1115–1150, December 1989.

[49] Shekita, E., and M. Zwilling, "Cricket: A Mapped, Persistent Object Store", *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, MA, September 1990.

[50] Singhal, V., S. V. Kakkad, and P. R. Wilson, "Texas: An Efficient, Portable Persistent Store", *Proceedings of the Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[51] Soley, R. M., and W. Kent, "The OMG Object Model", *in* [27].

[52] Stonebraker, M., "User Interfaces and Embedded Programming", introductory section to chapter 9 of [54].

[53] Stonebraker, M., "New Data Models", introductory section to chapter 10 of [54].

[54] Stonebraker, M. (ed.), *Readings in Database Systems*, Morgan Kaufmann, San Francisco, CA, 1994.

[55] Sun Microsystems, *The Java Language Specification*, Sun Microsystems, Inc., Mountain View, CA, 1995.

[56] Thomas, G., and R. van der Linden, *Remote Database Queries in Open Distributed Systems*, Technical Report APM.1138.01, ANSA, Cambridge, UK, 1994.

[57] Versant Object Technologies, *Versant Technical Overview*, Versant Object Technologies, Inc., Menlo Park, CA, 1990.

[58] White, S. J., *Pointer Swizzling Techniques for Object-Oriented Database Systems*, Ph.D. Thesis, University of Wisconsin–Madison, 1994.

[59] Wilson, P. R., and S. V. Kakkad, "Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware", *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems)*, Paris, France, September 1992, pp. 364–377.

[60] X/Open Company, *Distributed Transaction Processing: The XA Specification*, X/Open Document C193, X/Open Company Ltd., Reading, UK.

[61] Zdonik, S. B., and Maier, D. (eds.), *Reading in Object-Oriented Database Systems*, Morgan Kaufmann, San Francisco, CA, 1990.

[62] Zdonik, S. B., and Maier, D., "Fundamentals of Object-Oriented Databases", opening chapter of [61].