The O_2 Object Manager: an Overview

Fernando Velez, Guy Bernard and Vineeta Darnis
Altaïr BP 105, 78153 Le Chesnay Cedex France

Abstract

O_2 is the Object-Oriented Database System being developed at Altaïr. The O_2 Object Manager is the layer of the system concerned with complex object access and manipulation, transaction management, persistence and disk management, and distribution in the server/workstation configuration. This paper describes the main choices made in the design of the Object Manager, and presents its overall architecture.

1 Introduction

The major objective of Altaïr is to build a new generation development environment for data intensive applications. The functionality of the system should include that of a DBMS, of a programming language, and of a programming environment. The target applications of the system are business applications, transactional applications (except very high performance transaction processing systems), office automation, and multimedia applications. The physical configuration we aim at consists of a server connected to a set of workstations (which may be heterogeneous). The server is the common repository for shared data. The target customers are application programmers and end users. Our main interest is in application programmers, since we consider the major problem to be the improvement of programmer productivity.

To meet these requirements, we decided to build an object-oriented database system and its programming environment. O_2 is both a Database System and an Object-Oriented System. As a Database System it provides support for accessing and updating large amounts of persistent, reliable, and shared data. As an Object-Oriented system, it supports features such as complex objects with identity, inheritance (of classes or types), encapsulation (of an object state by the methods defined on its class), overriding (redefining methods in classes), and run-time binding of methods to objects. Our interpretation of these notions are condensed in the definition of the O_2 data model [22], [21]. Section 2 presents a quick overview of the model.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the Fifteenth International Conference on Very Large Data Bases

O_2 has a complete programming environment. It provides tools such as editors, browsers, and debuggers. When developing an application, programmers specify a schema and write the code for methods. The schema is specified in the O_2 language. Methods are written using standard programming languages. In the current implementation of the system, methods are written in either CO_2 or *BasicO*$_2$, which are extensions of C and Basic respectively. The schema declarations are interpreted and handled by the Schema Manager. Methods are precompiled, then compiled by the "host" programming language compiler, and they may be loaded and executed.

A first throwaway prototype was built in 1987 and is reported in [3]. Since then, we redefined the system at all its levels: data model and language, programming environment, compilers, Schema Manager and Object Manager. The work reported here concerns only the last module.

The Object Manager (OM) is the piece of software that handles persistent and temporary complex objects with identity. Furthermore, objects are shared, reliable and move from a workstation to the server and vice versa. The OM is used by all the upper modules of the system.

We believe that the OM synthesizes techniques proposed in the database field and in the object-oriented programming field. A similar approach is found in GemStone [24], Orion [19], Encore [14], Vbase [2], and Iris [11]. However, it is is original in the following respects. Firstly, in the way it handles distribution: applications may run either entirely on the server, or entirely on the workstation, or the programmer may migrate control from one machine to another at his will when passing a message to an object. We consider this to be a powerful performance tuning facility, not found in the systems mentioned above and well adapted to a server/workstation configuration (the first three fall in this category).

Secondly, the system adapts itself to a wide range of application requirements ranging from development of applications to execution of data-intensive applications. We distinguish a *development mode* in which users are programmers developing applications from an *execution mode* in which previously built applications are executed and the main concern is performance. The systems mentioned above do not make such a difference and run in a single mode.

Lastly, the system implements a number of facilities that make the life easy for application programmers: (i) exceptional attributes for tuple objects may be attached

Amsterdam, 1989

at any time without performance degradation, (ii) persistence is implemented with a simple composition-based schema in which deletions are implicit (this obviously is not new in the programming language world), and (iii) clustering issues are clearly separated from the schema information and specified by the DBA in the form of a subset of the composition graph.

The rest of this paper is organized as follows: Section 2 discusses the functional requirements of the OM both with respect to the O_2 data model and with respect to the applications the system is intended to support. Section 3 presents the main design choices we faced when developing the OM. Section 4 presents an architectural overview of the system. Section 5 compares our system to other related work, and Section 6 concludes the paper, summarizes the differences with the first prototype reported in [3] and presents some future extensions.

2 Functional Requirements

2.1 Data Model Requirements

In what follows, we present the particularities of the model pertinent to the system design. In O_2, we distinguish between *values*, on which we can perform any of a set of predefined primitives, and *objects* which have identity and encapsulate values and user defined *methods*. Values can be *set*, *list* or *tuple* structured, or *atomic*. Each value has a *type* which describes its structure. Objects belong to *classes*. A class has a name, a type t (specifying the common structure of the set of objects of the class) and a set of methods M. Every object of a certain class has a value of type t and has the same set of methods M.

Classes and types are partially ordered according to an inheritance relationship. Multiple inheritance is supported. Our interpretation of tuple types follows the one proposed by Cardelli in [7], implying that inheritance has natural set inclusion semantics: if C is a subclass of C , then the set of all possible instances of C is included in the set of all possible instances of C . The prescriptive interpretation of tuple types implies that a tuple object or value may have "exceptional" attributes, i.e. attribute values not declared in its class or type. The O_2 code below shows this:

```
   class Person type tuple (name:string,
age:integer);

   O2 Person x ;
   *x = tuple(name: ''john'', age: 27,
my_opinion: ''nice fellow'') ;
```

Here, the expression "*x" refers to the value encapsulated in object x. Reference to values encapsulated in objects is allowed only in methods of class `Person` (or in other methods if the de-encapsulation method "*" is exported by the creator of class `Person` - see [23]). In order that the `my_opinion` attribute could be used while conserving a "strict" typing, it is necessary to have in the language a clause which permits to verify the existence of this exceptional attribute. This is scheduled for the next version of the language.

A *named* object or value is an object or value with a user-defined name. These objects or values are the roots of persistence. Persistence is *user controlled*: in order to make an object persistent, the user has to make it a component of an already persistent object.

An O_2 *schema* consists of classes, types, named objects and named values. A query language has been designed [4] which uses the distinction between objects and values and the existence, in the data manipulation language, of primitives to manipulate structured values.

2.2 Application Requirements

The system should support a wide range of application requirements. Consider the three following utilization scenarios:

(i) Users are programmers developing O_2 applications. They constantly define new classes and new methods, and methods are generally tested with few objects in main memory.

(ii) Users are "end users" (i.e., not necessarily programmers), and they execute applications which have been previously developed and tested. Here, we have a notion of "application" which contains a set of classes and methods. In this mode, the essential concern is execution speed. Data are massive, shared and have to be reliable. The schema is static.

(iii) Users are programmers developing (rapid) prototypes with O_2. Data would fit in main memory in many cases and is seldom shared.. Execution speed is important and the schema may be flexible.

These three scenarios imply very different and contradictory requirements. In the first scenario, which we call *development mode*, flexibility is at a premium, and therefore, it is important to optimize method compilations and schema concurrency protocols. Execution speed is less important.

In the second scenario, the *execution mode*, we need all the functionality of a database system: persistence, disk management (i.e., indexing, clustering, and smart buffering), concurrency on objects (not necessarily on the schema), and recovery. Message passing should be optimized, referencing an attribute of a tuple value should be done in "compiled mode" (i.e., the system should not interpret an attribute name), and operations on large sets and lists should be done efficiently.

In the third scenario, the *resident mode*, the system could load all persistent data in main memory to enhance execution speed. This is in contrast with what we could call *object fault mode* which is typical of Data Managers.

2.3 Architectural Requirements

The OM should be as *canonical* as possible, i.e., other projects should be able to use the OM as a back-end in charge of persistence and transaction management for complex objects. This idea is not new: other projects such as Mneme [26] and ObServer [14] pursue this idea of a canonical object manager.

Another requirement is that it should be easy to downgrade the system to a single-site machine.

We tried to use existing products as much as possible. We use WiSS, the Wisconsin Storage System [10], as the low-level layer of the OM to provide persistence, disk management and concurrency control. It runs under Unix[1] System V but bypasses the Unix File System and does its own buffering.

3 Main Design Choices

3.1 Modes of Operation

3.1.1 Compiling Modes

The sharp differences in application requirements between the two modes (development mode vs. execution mode) mentioned in the previous section drove us to build two different "compiling modes". We designed the OM in such a way that these modes affect only the compilers. They do not imply building two different versions of the OM. In any case, both modes must share the same persistent data.

Execution mode applications are the result of "deeply compiling" some classes and methods developed by a programmer, to gain speed when executing these methods. The code is optimized in the following respects:

(i) Late binding is replaced by a function call whenever possible.

(ii) Access to tuple attributes by name is replaced by physical offsets.

(iii) No run-time method fault occurs: methods are loaded statically. In development mode, methods are loaded dynamically, as flexibility is required.

3.1.2 Transaction Modes

The OM interfaces either with a full-fledged execution mode application or with a development mode *session*. We shall denote both by the term *application*. An application is composed of one or more *transactions*.

Our notion of transaction differs from the classical database notion of transaction (namely, an atomic and serializable sequence of database commands) in the following respects:

(i) We distinguish concurrency on schema from concurrency on objects, and allow them to be activated or disactivated independently.

(ii) Recovery may be switched on or off. When running prototype applications, one may be willing to sacrifice safety for the sake of performance.

(iii) A transaction may choose to run in resident mode or object fault mode. Note that this is specified at execution time, not at compile time. A good candidate to run in resident mode is the Schema Manager. In the current version, the Schema Manager can run in object fault mode if it wishes to.

3.2 Handling Distribution

The OM has a *workstation version* and a *server version*. Both versions have (almost) the same interface. The main distinction is in the actual implementation: the workstation version is single user (as a workstation is single-user), memory based; while the server version is multi-user, disk based.

An important problem we faced while designing the system was: given an (execution mode) application program in O_2, how do we decide which part runs on the server and which part runs on the workstation? The main goal is to (i) make tasks run efficiently in the system configuration, and (ii) to program distribution as simply as possible. Recall that we are not dealing with a distributed query problem, but with general programs, since methods are programs.

Several approaches can be taken. The simplest is to make distribution transparent to the application programmer and make everything run on a single machine. Making distribution transparent and letting the system determine the "best site" is an unsolved problem up to now.

We decided to make the distributed architecture visible to the application programmer (but not to the end user). The programmer will be aware of the existence of two machines (the server and the workstation) and may explicitly specify on which of the machines a message passing expression is to be executed.

```
                                    ...
    /* programmer specifies that the message
''filter'', which tests */
    /* each element of set ''my_set'', should
be run on the server */

        result_object = [my_set filter() on
server] ;
                                    ...
```

In this case, if the selectivity ratio is high and the set is large, running the method on the server may result in better performance. On the contrary, methods displaying an object, editing a document object or performing computations on few objects should be run on

[1] Unix is a registered trademark of AT&T laboratories

the workstation to offload the server. Note that we associate location with message passing and not with methods.

For messages without site specification, the system decides the evaluation site. Possible strategies include naive ones, such as "execute in the machine in which you are currently executing". This is the one we have implemented. See Section 6 for foreseen improvements to this strategy.

Our design choices were made according to the criteria of *simplicity*, *transparency*, *performance*, and *reliability*.

Simplicity (and thus portability) is achieved by choosing standard and well-proven tools: the local area network which links the workstations and the server is Ethernet, the transport protocols are TCP/IP. *Transparency* results from the fact that the end user is never concerned with either the client/server task distribution or the heterogeneity of the machines. *Performance* is obtained by execution migration and by designing a communication protocol optimized for object moves, which are the main potential bottlenecks of any client/server architecture. Lastly, *reliability* is obtained by a failure detection mechanism which prevents indefinite resource holding which might result from an abnormal process termination.

3.3 Object Access

Objects are uniquely identified and accessed by object identifiers (*oids*). Object identifiers could be "logical", i.e., give no information about their location in secondary memory, as in GemStone [24], Orion [19] and ObServer [14]. With logical identifiers a correspondence table between oids and physical addresses is needed. Moving objects in secondary memory is straightforward, but the object table might be very big as it would contain one entry for each object in the database. One disk access is likely to be performed to retrieve the object table entry of the object, and a second to retrieve the object. We have chosen the (persistent) identifiers to be physical identifiers, i.e., reflecting their location on disk. The choice is largely motivated by the aforementioned performance reasons. Roughly speaking, an object will be stored in a WiSS record and the object identifier will be the record's identifier, i.e., an RID.

A major problem with physical identifiers is moving objects on disk without changing their identifiers[2]. The solution we adopted is to use forwarding markers (as in relational data managers such as System R). We modified WiSS in order to make the Object Manager have direct control over this mechanism.

[2]If we were to change the identifier of an object o we want to move on disk, we should be able to attain all objects referencing o to update their references, but this would imply using backward references in the composition hierarchy, and we consider these too heavy to maintain.

3.4 Object Representation

Recall that the O_2 model distinguishes objects from values (Section 2.1). In the OM, we deal only with objects and atomic values. Structured values will be given an identifier and are managed as "standard" objects. The system supports both the primitives for manipulating values as well as the message passing mechanism for objects. At the language level, objects are distinguished from values and errors such as passing a message to a value or applying a primitive to an object will be trapped at compile time. In the OM, however, there are primitives which distinguish oids denoting objects from oids denoting values.

3.4.1 Tuples

On disk, a tuple is represented as a record stored in a page. When a tuple outgrows a disk page, we switch to a different representation suitable for storing long records, the Long Data Item (or LDI) format provided by WISS. The oid of the tuple is unchanged (the RID of the original record).

In main memory, tuples are represented as a contiguous chunk containing the actual values. Only strings are stored away from the main chunk, which contains pointers to the proper locations. This way the strings may grow or shrink without requiring the entire object to change location. An exception to this rule comes from the fact that in O_2, a tuple object may have "exceptional" attributes, i.e., attribute values not declared in its class (Section 2.1). In such cases, the tuple object may grow in length. When a tuple grows, if an in-place extension is not possible, a level of indirection for the entire tuple value is generated.

3.4.2 Lists

Lists, which can be more accurately called ***insertable arrays***, are represented as ordered trees as in [31] (with slight modifications to make fast scans). An ordered tree is a kind of B-tree in which each internal node contains a count of the nodes under it. The insertion and deletion procedures have to update node-counts (this is the essential difference from standard B-tree management). This structure is efficient to store small and large lists. In EXODUS [8], it has been used to implement long data items.

3.4.3 Sets

The representation for large sets of objects needs to be such that (i) membership tests are efficient, and (ii) scanning the elements of the set is also efficient. A set of objects is itself an object containing object identifiers of its members (the objects themselved are stored according to clustering strategies described in section 3.5). WiSS provides two kinds of indices: hash indices and B-trees. We use B-tree indices to represent large sets. However,

using an index for a small set would be too costly. Therefore, there is a limit under which a set is represented as a WiSS record; a convenient value for this limit is the maximum record size in WiSS. Small sets are kept ordered. This decision was motivated by the fact that large sets are kept ordered, and binary operations on sets take advantage of this uniformity: unions, intersections and differences are programmed using "merge" algorithms.

3.4.4 Multimedia Objects

Two types of multimedia objects are implemented: unstructured text and Bitmap. From the user point of view, they are instances of the predefined classes **`Text`** and **`Bitmap`**. The predefined methods in these classes are **`display`** and **`edit`**. From the system point of view, texts are atomic objects of type **`string`** and bitmaps are atomic objects of type **`bytes`**, an unstructured byte string preceded by its length.

3.5 Persistence and Clustering on Disk

Recall that persistence is defined in the O_2 model as reachability from persistent root objects, which are named objects or values (Section 2.1). This is implemented by associating with each object a reference count. An object persists as long as this counter is greater than 0.

Newly created, persistent objects are given a persistent identifier when they are inserted in a file at transaction commit. The mapping of objects to files depends on control data given by the database administrator describing the placement of objects: the Placement Trees.

Placement trees are described extensively in [6]. The main idea is that if several objects are used together frequently, we should put them as close to one another as possible on disk. The main heuristic used here to postulate that two objects will be used together frequently is their relationship through composition structure. Roughly speaking, if class C' is a son of class C in a placement tree and an object o' of class C' is a component of object o of class C, we try in fact to store o' as close as possible to o. To do so, we use the elementary clustering facility offered by WiSS.

3.6 Handling concurrency at the O_2 level

Concurrency on objects is handled by WiSS at the server. However, the distributed architecture of the system does not make life easy. Consider a transaction executing on the workstation and updating shared objects. Before reaching the workstation, objects are read-locked. Note that it is difficult to anticipate which objects need to be write-locked, as objects are operated by methods of arbitrary complexity. There is no concurrency control on the workstation, as workstations are considered to be single-user. The problem is to ensure consistency, and this in a cheap way. Straightforward solutions are (i) each time the transaction wants to update an object in the workstation, migrate execution to the server and perform the update there: the concurrency control mechanism of WiSS will ensure consistency, (ii) update the objects locally and maintain an update log that will be executed in the server at transaction commit, (iii) run the transaction in the workstation and in the server in parallel (this is less straightforward!). Solution (i) has obvious performance problems, solution (ii) seems complicated to implement and has contention problems as all controls are performed at commit time, and solution (iii) is difficult to implement because the transaction itself may migrate execution site from one machine to another (see Section 3.2). None of these being all too satisfactory, we chose the solution now presented.

Concurrency in WiSS is handled by a two-phase locking algorithm on pages (and files). WiSS has an internal **`lock_page`** primitive which sets locks on pages. Our solution is to move this primitive to the programming interface, and ask for write locks explicitly from the workstation before an object is updated. The update proceeds in the workstation asynchronously without waiting for an answer. Unless the answer is **`rollback`**, the workstation is not informed. Otherwise, the process running the transaction in the workstation is informed and the transaction aborted. At transaction commit, the workstation process asks the server process if all requested write locks have been granted (we call this "pre-commit"), and only if this is the case, are the objects transferred. This way, consistency of shared objects in the server is preserved.

As mentioned previously, we handle concurrency on the schema differently from concurrency on objects. This is a consequence of our application requirements summarized in Section 2.2, but it is also due to the following observation: in development mode, the schema *is* the hot spot of the system. Even executing a message passing expression implies reading the schema information because methods belong to the schema. Therefore, a custom concurrency control for the schema is being designed which takes into account semantic information about schema updates to allow for increased parallelism. It just does not seem wise to put a read lock on a page each time a message passing expression is solved.

4 Overview of the System

4.1 Global Architecture

The OM is divided into four layers: (i) a layer which copes with the manipulation of O_2 objects and values and transaction control, (ii) a Memory Management layer, (iii) a Communication layer taking into account object transfers, execution migration, and application downloading, and (iv) on the server, a Storage layer devoted to persistence, disk management, and transaction support implemented by WiSS.

The following process layout has been adopted: on the workstation, an application and the workstation version of the OM form one unique process. There will be

as many processes as running applications.

For each process running on a workstation, there will be a *mirror* process running on the server. In addition, there may be some *terminal* application processes running on the server which don't have any corresponding "partner" on a workstation. Given the large number of such concurrent applications on the server, the OM is compiled as re-entrant library modules to be shared among all applications. The lock table and the buffer managed by WiSS are shared by all processes. The OM Object Memory is also a global buffer in shared memory, as detailed below.

Both versions of the OM (workstation and server) are illustrated in Figure 1.

4.2 System Interface Module

This layer is the gateway to the OM. All object manipulations done by a transaction are treated by this module.

4.2.1 Object Manipulation

This submodule performs the following tasks:

(i) Creation and deletion of structured types (creation of classes and methods is the responsibility of the Schema Manager and not of the Object Manager).

(ii) Creation and deletion of objects.

(iii) Retrieval of objects by name.

(iv) Support for the predefined methods for objects. These are the methods of predefined class `Object`, namely: `value_equal`, `deep_equal`, `value_copy`, `deep_copy`, `display`, and `edit`.

(v) Support for set, list, and tuple objects.

For sets and lists, we have implemented a two-layer structure in which the upper layer takes care of logical operations (and reference counts) and the lower layer handles the data structures.

4.2.2 Support for Late Binding

This submodule exists both on the workstation and the server. In development mode, it provides support for late binding and also handles the application of the selected binary code to the receiver object. In execution mode, the message is replaced by a function call whenever possible and in the few cases which remain unsolved, method name resolution is done by the executing code in an ad-hoc manner.

4.3 Transaction Support

An O_2 transaction maps directly to a WiSS transaction. This submodule supports the functionality described in Section 3.1.2 and handles concurrency according to the strategy presented in Section 3.6.

When a transaction runs in memory resident mode, the system is given information about the persistent root objects it accesses, and fetches all objects accessible from these at transaction start. Once this is done, all inter-object references are changed to memory addresses.

At both sites, this submodule keeps track of the set of dirty objects (this includes newly created objects) and the set of persistent objects to be deleted from the database. When control is transferred from one site to another (at transaction commit or when migrating execution), these two sets, as well as the objects referenced to by these sets, are also transferred. In this way, transactions preserve their execution context. The coupling of the workstation version and the server version of the system is tight, as they communicate via "lower" system interfaces, as it is shown in the following sections (this has also been done in AIM-P [16]). This contrasts with (relational) distributed database systems, in which communication is done mostly via the high-level relational database interface.

4.4 Memory Management Module

This layer takes care of translating object identifiers into memory addresses. This includes handling object faults for objects requested by the application and not currently in memory. It is also responsible for managing the space occupied by objects in main memory.

As in Orion [19] and GemStone [24] (but unlike ObServer [14]), a dual buffer management scheme is implemented: a page buffer implemented by WiSS and an object buffer pool, the ***object memory***. Objects in the page buffer are in their disk format. In the object memory, they are in their memory format.

On the server, an object fault implies reading a WiSS record and transferring it between the page buffer and the server object memory. Even though an object corresponds to one WiSS record, on every object fault, all the valid records on the same page as the object in question are transferred into the object memory. This "read-ahead" strategy is based on the fact that objects which have a strong correlation between them are clustered on the same or nearby pages and reading an entire page (which is anyway placed in the page buffer by WiSS) will accelerate further processing.

As some degree of sharing is expected among applications, the server object memory is implemented as a data segment shared by all the concurrent processes [15].

On the workstation, the object memory is private to each application. The memory allocation and deallocation tasks are left to the Unix virtual memory mechanism. An object fault is addressed to the Communication Manager, which in turn asks the server mirror process to send the object across the network. As before, all objects stored in the same page as the requested object are transferred to the workstation.

Both on the server and on the workstations, the memory address at which an object is stored never changes until the object migrates to another machine

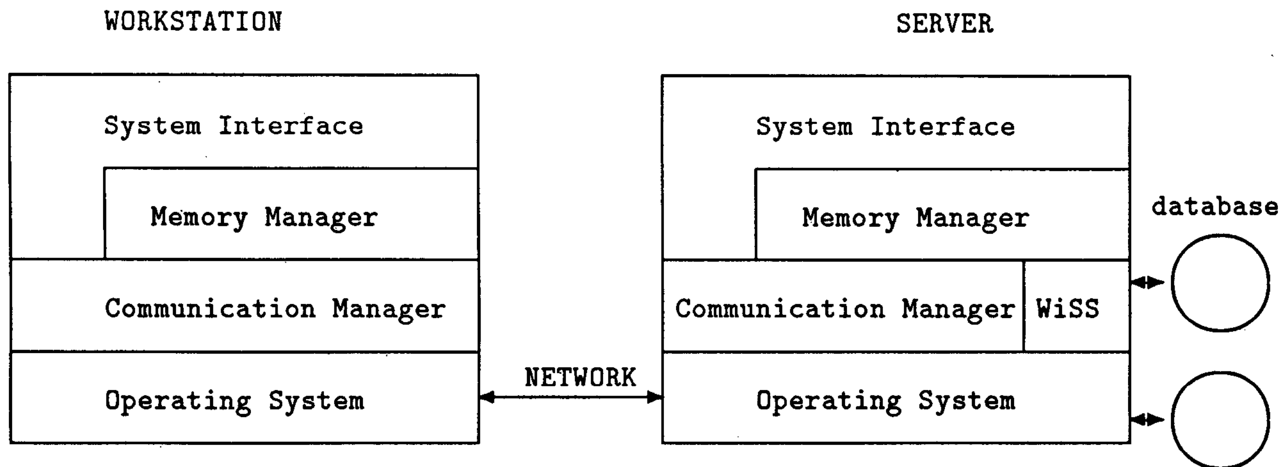

Figure 1: Architecture of the OM

or is written out to disk. While an object is in memory, in order to access it given its identifier, we use an object table. This table is hashed on identifier values and contains entries for resident objects only, as in LOOM [18].

4.5 Communication Module

4.5.1 Session Level

When an application is started on a workstation, a *mirror application process* must be started on the server in order to interact with the lower layers of the system.

The *workstation application process* and the *mirror application process* behave differently: whereas the workstation application process executes the code of the application as a transaction, the *mirror application process* is passive: it goes to sleep until receipt of an incoming message, processes it by *upcalling* the appropriate primitive and goes back to sleep until the next message arrives, or the workstation application process is ended.

4.5.2 Presentation Level

The heterogeneity between machines is handled by a presentation protocol, namely *XDR* [1], chosen for its simplicity. For object moves, the size of an object when received at a site depends on the machine parameters. In order to minimize extraneous memory-to-memory copying, this size is computed from the type of the transferred object. It is used to determine the place in memory where the object data will be copied, and data transfer is done directly from network to this memory address. Another optimization consists of grouping in a single message several objects to move (for instance at commit time). This minimizes the number of network accesses.

4.5.3 Execution Migration

In the current version of the system, the execution site for a message passing is specified statically by the application programmer. Embedded execution transfers may appear. For instance, if a selection operation results in displaying one tuple over 1000 in average, the programmer may specify that the selection operation should be done on the server but the display, of course, can be done only on the workstation. This way, the two sites may act alternatively as client or server. Our execution migration protocol is thus more than an unidirectional classical Remote Procedure Call mechanism.

5 Comparison with Related Work

We attempt to compare here the O_2 OM with other existing object managers. Caveat: as there is a lack of agreement on higher level object-oriented concepts for databases, object managers actually implement different kinds of functionality for different kinds of data models.

A major concern throughout the design has been to build a common kernel for different object oriented paradigms. Outside Altaïr, the OM is currently being used as a back-end bearing persistence to a frame-based expert system generator [27]. In fact, most of the semantics of the O_2 type system has been shifted to the upper layers, i.e., the Schema Manager and the Compilers. The part of the OM depending on classes and methods is essentially the message-passing mechanism and has been clearly isolated from the rest of the system, which handles shared and persistent typed objects with identity. Most object-oriented languages propose tuple objects and some have a notion of (ordered or unordered) collections. In this sense, the OM implements "state of the art" complex objects. The difference with nested relational kernels such as DASDBS [29] is that the latter are geared towards optimizing selections and joins,

and don't support traversing the composition graph, nor important (object-oriented) functionality such as orthogonal persistence, garbage collection or support for methods and message passing.

Other projects such as Mneme [26] and ObServer [14] pursue this idea of a "canonical" object manager. Both provide untyped objects, the typed layer being implemented by a *client* of the object manager. In this case, traversing the composition graph is either done at the client level, with an obvious performance penalty, or the object manager must distinguish between object references and other kinds of data within the object, as in [26].

For space reasons, we will restrict ourselves to a subset of object-oriented DBMSs, namely GemStone [24] and Orion [19]. Other well known systems are Vbase [2], Iris [11]. Extensible DBMS such as Exodus [9] and Postgres [30] are close to object-oriented DBMS for their ability to deal with procedures and with an inheritance hierarchy.

5.1 GemStone

The original objective of GemStone [24] was to make Smalltalk a database system. Stone, the lower layer of GemStone has the same kind of functionnality as the OM : it provides persistence, secondary storage management, transactions, concurrency control, recovery, support for associative access, and authorization functions. However, the decomposition in processes is different than in O_2, as Stone is run as a single process on the server, and it communicates with different upper layer processes (Gem processes), one for each active user, also running on the server. Gem corresponds, roughly speaking, to the virtual image of the standard Smalltalk implementation. On the workstation, an interface to GemStone (PIM) has been developed and implements remote procedure calls to functions supplied by GemStone. So, the way the client/server architecture is handled is completely different as in our system.

Applications run exclusively in "development mode" in GemStone. Also, transaction management in GemStone bears little similarity to ours. An optimistic concurrency control scheme is implemented. Access conflicts are checked at commit time, rather than preventing them through locking. The main disadvantage with this scheme is that a long transaction can loose arbitrary amounts of work if prevented from committing. To alleviate this problem, a pessimistic concurrency control based on locking is explored in [28].

One of the major innovations of GemStone is its indexing mechanism to support associative access on large collections [25]. The main feature is that indexes can be defined on a *path* along the composition hierarchy, and that they index into sets rather than into classes.

5.2 Orion

Orion [5], [19] is an object-oriented DBMS developed at MCC extending Common-Lisp with object-oriented programming and database capabilities. The database type system is different from the host language type system, and Orion's persistence is not orthogonal. A version of Orion implements a multi-user, multi-task system in which a server provides persistence and sharing of objects on behalf of several workstations. Unlike O_2, an application does not have the ability to execute general methods in the server and the workstation while it runs. As with GemStone, the system is geared towards development mode.

The architecture of Orion consists of the following modules: (i) a *message handler* receives all messages sent to Orion objects, (ii) an *object subsystem* provides functions such as schema evolution, version control, query optimization, and multimedia information management, (iii) a *transaction subsystem* provides concurrency control and recovery mechanisms, and (iv) a *storage subsystem* provides access to objects on disk. The two last modules implement functionalities similar to those of the OM: The storage subsystem implements a dual buffer management scheme (page buffer and object buffer) and maintains a resident object table hashed by oid. However, as oids are logical, there is another hash-based oid-to-physical-id for all objects in the database. Buffer management for objects in the object buffer is far more complex than ours: fragmentation is dealt with by a garbage collection mechanism that reclaims space occupied by objects that can still be referenced to by the application, and thus an intermediate structure, the *resident object descriptor* or ROD, has to be introduced between the resident object table and the actual object. Applications point to RODs only, and these may not be swapped.

Transaction management [12] is rather "classical": concurrency control is based on a locking protocol and recovery is based on logging. However, a composite object (a hierarchy of exclusive and dependent composite objects) may be locked as a whole. To this end, the hierarchical locking protocol [13] has been extended. Also, a class-lattice locking protocol, which is needed to allow access to instances of a class while preventing changes to the definitions of the superclasses of the class has been proposed. Other important database features in Orion are indexing, authorization and version support. Indexes may be defined on a class or on subclasses of a class, and are used when evaluating queries.

6 Conclusions and Future Work

In this paper, we described the main choices involved in the design of the O_2 Object Manager, and presented its overall architecture. The OM is used by all the upper modules of the O_2 system. It combines well known techniques from the database field and the object-oriented programming field, and has the following original as-

pects:

(i) The programmer has control over distribution. This is done in an easy way when passing messages to objects. We consider this to be a powerful performance tuning facility.

(ii) The system adapts itself to a wide range of application requirements ranging from development of applications to execution of data-intensive applications.

(iii) The system implements a number of facilities that make life easier for application programmers: (i) exceptional attributes for tuple objects may be attached at any time without performance degradation, (ii) persistence is implemented with a simple composition-based schema in which deletions are implicit, and (iii) clustering issues are clearly separated from the schema information and specified by the DBA in the form of a subset of the composition graph.

Our first prototype [3] implemented an early version of the data model reported in [22]. There were no notion of values and lists were inexistent. All sets were implemented in the same way, regardless their size. Persistence was attached to classes, not to objects, so all objects of a persistent class were persistent. There were no clustering strategies, no transaction modes, no difference between execution mode and development mode. Finally, the Schema Manager was built in an ad-hoc way without using the object manager. For the next version of the system, the major thrust will be in the following directions :

(i) An ad hoc concurrency control for the schema, as mentioned in Section 3.6, is under design and will be implemented shortly.

(ii) Recovery and rollbacks are not implemented in the current version of WiSS. We plan to add these shortly. To support long transactions, we also plan to provide "savepoints" in order not to loose arbitrary amounts of work.

(iii) Indices will be provided at the O_2 level (WiSS provides already B+ trees and hash-based indices on files). The relationship of indices with both the inheritance hierarchy and the composition hierarchy (which has been explored separately in [20] and [25] respectively) is currently being investigated.

(iv) Our large objects are currently limited by the size of a WiSS long data item (1.6 Mbytes). We are looking into the possibility of modifying WiSS to be able to store large objects with virtually no size limitation.

(v) More work is needed towards determining when objects are "badly clustered" and should be moved on disk. Also, an incremental restructuring policy should be used.

(vi) For messages without site specification, the system currently implements message passing in the same site in which it runs. This strategy can clearly be improved by letting the system decide dynamically on the site of application of the method.

(vii) Versioning and authorization will be taken into account in the next version of the system.

Acknowledgements

We thank François Bancilhon for comments on an earlier draft of this paper and for the many discussions we had during the design of the system. We also thank the following persons involved in the development of the Object Manager: Véronique Benzaken, Constance Bullier, Philippe Futtersack, Gilbert Harrus, John Ioannidis, Jean-Marie Larchevêque, and Dominique Steve. The OM also benefit from discussions with Christophe Lécluse and Philippe Richard and especially from those with Sophie Gamerman and Claude Delobel. Paris Kanellakis, Michel Scholl and John Ioannidis read earlier drafts of this paper and suggested many improvements.

References

[1] *"External Data Representation Protocol Specification"*. SUN Microsystems, February 1986.

[2] T. Andrews and C. Harris. "Combining Language and Database Advances in an Object-Oriented Development Environment". In *Proceedings of the OOPSLA conference*, October 1987.

[3] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. "The Design and Implementation of O_2, an Object-Oriented Database System". In *Proceedings of the OODBS II Workshop*, Bad Munster, RFA, September 1988. Springer Verlag.

[4] Francois Bancilhon, Sophie Cluet, and Claude Delobel. *"Query Languages for Object-Oriented Database Systems: Analysis and a Proposal"*. Technical Report, Altaïr, 1989. In preparation.

[5] J. Banerjee et al. "Data Model Issues for Object-Oriented Applications". *ACM Transaction on Office Information Systems*, 5(1), April 1987.

[6] V. Benzaken and C. Delobel. *"Clustering Objects on Disk in an Object-Oriented Database"*. Technical Report, Altaïr, 1988. To appear.

[7] L. Cardelli. "A Semantics of Multiple Inheritance". *Computer Science*, Semantics of Data Types, Lecture Notes(173), 1984. Springer Verlag.

[8] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the exodus extensible database system. In *Proceedings of the 12th. VLDB Conference*, Kyoto, Japan, August 1986.

[9] Michael Carey et al. "The Architecture of the EXODUS Extensible DBMS". In *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[10] H.-T. Chou, David J. DeWitt, Randy H. Katz, and Anthony C. Klug. "Design and Implementation of the Wisconsin Storage System". *Software - Practice and Experience*, 15(10), October 1985.

[11] D. H. Fishman et al. "IRIS: an Object-Oriented DBMS". *ACM Transaction on Office Information Systems*, 5(1), January 1987.

[12] Jorge F. Garza and Won Kim. "Transaction Management in an Object-Oriented Database System". In *Proceedings of the ACM SIGMOD Conference*, Chicago, Illinois, May 1988.

[13] James N. Gray. "Notes on Database Operating Systems". In *Operating Systems: An Advanced Course*, Springer Verlag, 1978.

[14] Mark F. Hornick and Stanley B. Zdonik. "A Shared, Segmented Memory System for an Object-Oriented Database". *ACM Transaction on Office Information Systems*, 5(1), January 1987.

[15] John Ioannidis. *"Oh no!, we need Shared Memory"*. Technical Report, Altaïr, 1989. in preparation.

[16] k. Kuspert, P. Dadam, and J. Gumauer. Cooperative object buffer management in the advanced information management prototype. In *Proceedings of the 13th VLDB Conference*, Brigton, England, September 1987.

[17] T. Kaehler. "Virtual Memory on a Narrow Machine for an Object-Oriented Language". In *OOPSLA'86 Proceedings*, September 1986.

[18] T. Kaehler and G. Krasner. Loom - large object-oriented memory for smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.

[19] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F Garza, Darrel Woelk, and Jay Banerjee. "Integrating an Object-Oriented Programming System with a Database System". In *OOPSLA '88 Proceedings*, Los Angeles, California, September 1988.

[20] Won Kim, Kyung-Chang Kim, and Alfred Dale. *"Indexing Techniques for Object-Oriented Databases"*. Technical Report DB-134-87, MCC Technical Report, May 1987.

[21] C. Lécluse and P. Richard. "Modeling Complex Structures in Object-Oriented Databases". In *8th Symposium on Principles of Data Base Systems*, Philadelphia, Pennsylvania, March 1989. To appear.

[22] C. Lécluse, P. Richard, and F. Velez. "O_2, an Object-Oriented Data Model". In *Proceedings of the ACM SIGMOD Conference*, Chicago, Illinois, June 1988.

[23] Christophe Lécluse and Philippe Richard. *"The O_2 Database Programming Language"*. Technical Report, Altaïr, January 1989. Submitted for publication.

[24] David Maier and Jacob Stein. "Development and Implementation of an Object-Oriented DBMS". In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, MIT Press, Cambridge, Massachussets, 1987.

[25] David Maier and Jacob Stein. "Indexing in an Object-Oriented DBMS". In *International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

[26] J. Eliot B. Moss and S. Sinofsky. Managing persistent data with mneme: designing a reliable, shared object interface. In *Proceedings of the OODBS II Workshop*, Bad Munster, RFA, September 1988. Springer Verlag.

[27] Bertrand Neveu and Pierre Haren. "SMECI: an Expert System for Civil Engineering Design". In *First Int. Conference on Applications of Artifial Intelligence to Engineering Problems*, Southhampton, England, April 1986.

[28] Jason Penney, Jacob Stein, and David Maier. "Is the Disk Half Full or Half Empty?: Combining Optimistic and Pessimistic Concurrency Mechanisms in a Shared, Persistent Object Base". In *Workshop on Persistent Object Systems*, Appin, Scotland, August 1987.

[29] M. H. Scholl, H. B. Paul, and H. J. Schek. Supporting flat relations by a nested relational kernel. In *Proceedings of the 13th VLDB Conference*, Brigton, England, September 1987.

[30] M. Stonebraker. "The Design of the POSTGRES Storage System". In *Proceedings of the 13th VLDB conference*, Brighton, U.K., 1987.

[31] M. Stonebraker et al. "Document Processing in a Relational Database System". *ACM Transactions on Office Information Systems*, 1(2):143–158, April 1983.