WEA, A Distributed Object Manager based on a Workspace hierarchy

Didier Donsez, Philippe Homond, Pascal Faudemay

Laboratoire MASI / UPMC, 4 Place Jussieu, 75 252 Paris cedex 05, France
Internet : {donsez, homond, faudemay}@masi.ibp.fr

ABSTRACT

WEA is our implementation of a new architectural model for virtual memory access, the WorkSpace. It relies on a generalisation of client / server model and enables to build new distributed applications. The workspace supplies uniform access to a distributed persistent object store. This paper describes several ways of building multi-workspace architecture, and an implementation of this architecture. This implementation is based on new operating systems features.

Keyword Codes: C.2.4, D.1.5, D4.2
Keywords: Distributed Systems; Object-oriented Programming; Storage Management

1. INTRODUCTION

Object-oriented languages are a major advance in Software Engineering. However, the developer usually has to use file systems for data archival. File systems do not enable a simple memorization of pointers on the archive, or direct management of objects in the archive. Persistent Object Managers enable full and reliable integration of the functionalities of object-oriented programming languages, and of database systems.

Numerous Persistent Object Managers have been implemented [1-2] and commercialized [3-4]. However most of these Object Managers remain unsuitable to the need of new functionalities, or to performances equivalent to those of classical performance languages.

Among classical functionalities, full distribution on a local area network (LAN) implies access by each client to a large number of servers. Groupware implies the possibility for several transactions to see the modifications executed by each of them, and to choose among those modifications according to a common procedure. Multimedia applications must manage contiguous large objects, which can be accessed by adapted hardware (DMA).

An interesting approach to reach high performances is to entirely use low level mechanisms which are implemented by operating systems. This approach has been successfully started by ObjectStore [5]. This system manages mapping between objects identifiers [6], and physical addresses, using memory mapping mechanisms which install a file in an application address space.

New operating systems also enable application parallelism through Multithreading [7-8]. In this approach, simultaneous activities, called threads, share the address space of an application. Multithreading enables optimum use of arbitrary numbers of resources (processors, disks, communications links…). We consider that these mechanisms must be the basis for designing new Object Managers architectures.

Our research on a new type of Object Manager started with the criticism of an Object

Manager previously developed within our laboratory, VROOM version 1. VROOM implements a performance-oriented object manager with a client-server architecture, but does not use these system mechanisms.

The WEA system (WorkSpace Environment Architecture) uses memory mapping and multithreading to implement a new architectural model, the WorkSpace model. This model generalises the client-server architecture. It is based on a WorkSpace graph, where each WorkSpace (WS) can simultaneously be client and server.

Reflections on basic concepts of WEA have been worked out by a team distributed in MASI Laboratory and other laboratories. The WorkSpace model was first proposed by [9]. However, several implementations of this model are presently in study. In this paper, we present the WEA system, which is our implementation of the WorkSpace model.

In the following of this paper, we successively present the WorkSpace, several ways of assembling Workspaces, and the internal WorkSpace mechanisms in WEA.

2. FUNCTIONAL ASPECTS OF WORKSPACE

A WorkSpace is the local applications environment through which they share a database in a consistent and reliable way. For WEA, we define a framework for these applications, which is based on three models. The programming model is a specification of the object model, the execution model is based on transactional multi-threading, and the structural one distributes objects in a transparent way.

That's why we define an entity called the WorkSpace. In a classical database system, the WorkSpace is the space where a transaction memorizes or modifies its local data. These data are made visible to other transactions after their "commit" (end of transaction). In the WorkSpace architecture, the WorkSpace contains data which are directly or not visible by one or several transactions. The WorkSpace is a large virtual memory for persistent and temporary objects.

2.1. The WorkSpace as objects programming model

The description and manipulation language of the WorkSpace model is the C++ object oriented language [10]. WEA interface benefits from the power of this language, which has become a de facto standard in the market of object-oriented languages. Persistency is orthogonal to the object definition: any C++ object can be archived in a WEA database, whatever the complexity of its definition. The objects model is also orthogonal to objects instanciation: objects of a same class can either be created as persistent or temporary objects. A temporary object can also be made persistent at commit time.

The language is extended by generic structured types as arrays, sets, lists or dictionaries. These types are defined in a library which is delivered with WEA. This library proposes also large objets, i.e. very large byte strings specially dedicated to multimedia data types.

In WEA, objects are accessed via identifiers which are unique references in the whole distributed system, and are not re-allocated (except for large objects). Some objects can be named by the developer, using character strings, and are used as entry points in the database.

2.2. The WorkSpace as transactional model

Classical transactional model [11] is well adapted to the WorkSpace operation. In order to isolate modifications, the WorkSpace model defines a private WorkSpace for each transaction. Each transaction is executed by a thread coupled with this WorkSpace.

Parallelism can be introduced in an application by instanciating several simultaneous transactions. The application is no longer a list of sequential consultations and updates, but becomes a set of concurrent activities which communicate through sharing objects. Locking techniques of the transactional model offer a simple way to synchronize the access to the objects by concurrent threads of control or transactions.

2.3. The WorkSpace as structural model

The WorkSpace is also an entity which manages distribution at objects level (objects state and objects execution). The WorkSpace is a generic brick used to build distributed applications.

Database objects are distributed among volumes which are placed on different machines of the network. Each volume is exclusively controlled by a single WorkSpace which has two main characteristics : the accessibility of its private volumes by its transactions and the communication with other Workspaces.

The WorkSpace can publish its private volumes to other entities of the network. The other workspaces can subscribe to this WorkSpace volumes, in order to access pages of a volume in a transactional mode. In this case, subscriber workspaces are clients of the publisher. Thus using publish and subscribe, the WorkSpace unifies the classical notions of client and server. The publication-subscription mechanism is not reserved to the volume exportation. It is accessible to developers who want to write services according to the same connection principle. Publication and subscription are not limited to a single level: a client WorkSpace can publish the same service or another one.

In order to extend the WorkSpace possibilities, we define two types of behavior. The *passing WS* can be considered as a data and lock cache for its transactions and for the workspaces which subscribe to it. The *retaining WS* behaves as an including transaction with regard to transactions and subscriber Workspaces. Each of these transactions can work locally on a private context. Theses transactions can share their committed modifications without validating them to local or distant volumes. These modifications are globally validated to the database only when the WorkSpace commits. Combination of these two WorkSpace types enables to define and build application models, which are presented at the next section.

3. THE WORKSPACE HIERARCHY

The WorkSpace offers both functions of a server and of a client. This paragraph enlightens several ways of connecting Workspaces, in order to obtain either the classical client-server architectural model or the more recent cooperative one.

3.1. Client-server model

The present information system of an enterprise is built on the classical client-server model. In this model, each application executes a sequence of transactions on a client machine, and accesses archived objects which are served by server machines. Client-server architectures are designed straightforward. Three utilisations are considered. A WorkSpace operates as a server by publishing one or several volumes of the database. Clients are also subscriber Workspaces (fig. 1 WS A). There can also be an intermediary WorkSpace between a LAN and a WAN that plays the role of a concentrator and an objects cache for clients belonging to the same Local Network (fig. 1 WS B).

3.2. Model of application server

In an enterprise information system, the RPC model (Remote Procedure Call) may be used for protected considerations to limit the migration of non-relevant objects through the network. Some objects are too sensitive to be processed on a client. These objects must be executed by a specific service, which defines the limits of utilisation of these data, on a machine which is not one of the clients. In this case, the WorkSpace is an application server (fig. 1 WS D), which publishes services implemented by local transactions. The implementer will have the possibility either to use the basic communication mechanism (publication, connection, dialogue) used to implement the "object service", or to define another one.

3.3. Groupware and cooperative work model

Groupware implements the cooperation between several applications or users for the implementation of a common project. This type of cooperation may be controlled by a retaining WorkSpace which does not immediately validate the modifications brought by the subscribing transactions or client Workspaces (fig. 1 WS E). It is necessary to define a complementary mechanism to choose a common object version before the WorkSpace commits.

3.4. Standalone operation mode or private database

A user must be able to execute transactions on a private database (fig. 1 WS F), without any penalty resulting from pseudo-communications. In this case, there is a direct access to the database. The code and the binary file are the same as if the database was a distant one.

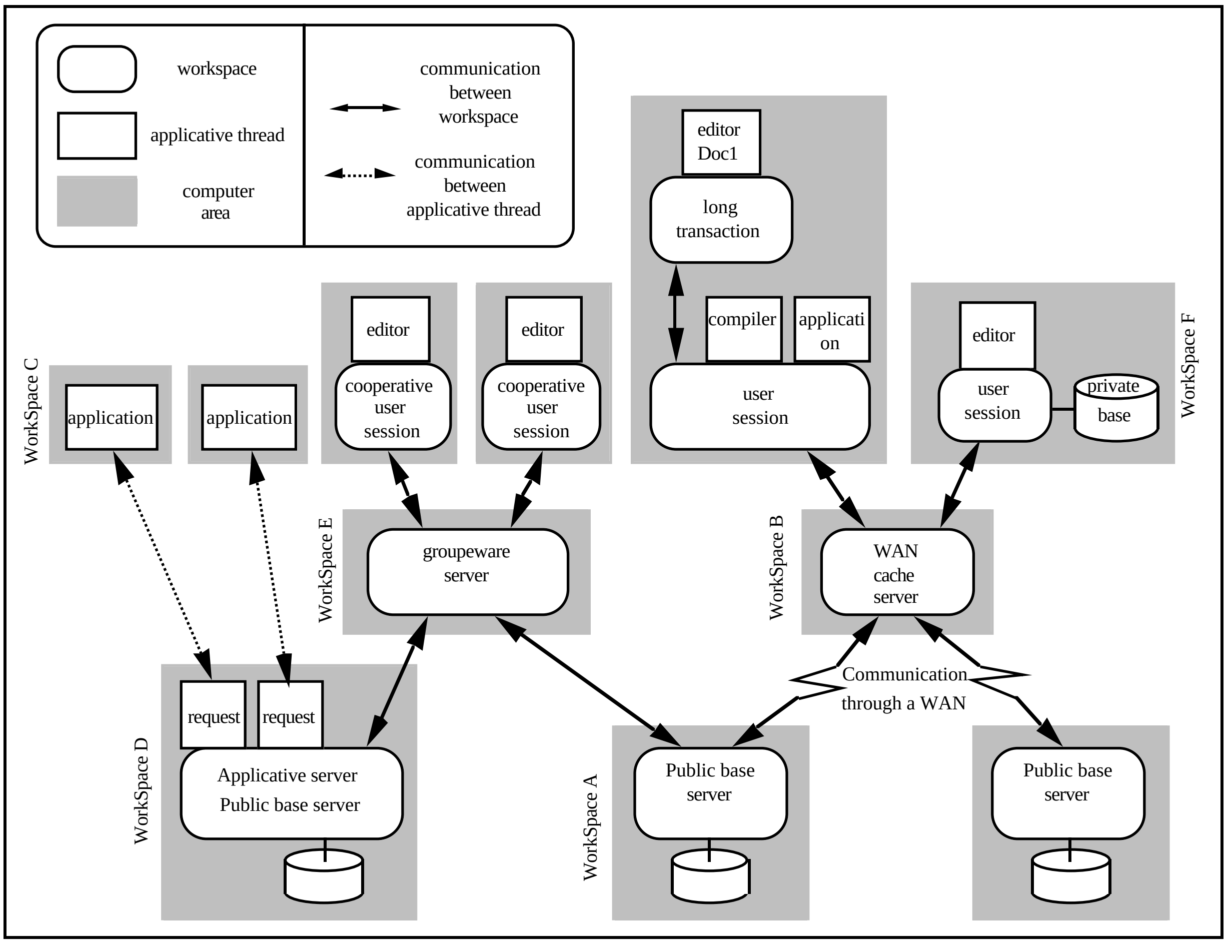

Figure 1 - WorkSpace Composition

4. WORKSPACE INTERNAL DESIGN

4.1. Multithreaded structure of the WorkSpace

The WorkSpace is a virtual memory address space of a process shared by several threads. The WorkSpace structure is designed to enable several transactions to be executed simultaneously, to ask objects to one or several server WorkSpace, and to serve clients WS in an asynchronous mode (fig. 2):

• The *User Thread* executes an application transaction. An application may instanciate several transactions simultaneously. Transactions also benefit from a common cache of objects pages loaded into the WorkSpace.

• Each pair of *Mux / Demux Threads* manages an asynchronous connection with a single server WS. This enables several User Threads to send several queries without serializing them by waiting the answers.

• Each pair of *Server Thread*s manages a connection with client Workspaces. The STin thread is in charge of receiving queries and the STout thread is in charge of answering them. They are the counterpart of Mux / Demux threads on the server, and operate in the same way, which results in an asynchronous dialogue between client and server.

The asynchronous operation mode of the WorkSpace is specially needed in the case of a WorkSpace hierarchy with several levels. In this case, when a request is not satisfied on an intermediate WorkSpace (§3.1), the request is transmitted to the inferior WorkSpace. While waiting for a request, other requests may arrived and can be processed.

4.2. The Page interface

The role of the Page interface is to install the image of the accessed pages in the WorkSpace address space. Modifications brought by a thread remain private to this thread until commit.

Memory mapping installs (or "maps") the image of a file (or of part of a file), in the address space of a process [12]. The access to the file can result from any memory access in the mapping area. Mapping may be shared, and in this case writes are directly executed on the file. Private mapping preserves the original image of the file, by executing the writes on a private copy of the modified pages (copy-on-write mechanism). The OS also enables to "protect" some areas of the address space from read or write access, and to detect them when they occur.

These mechanisms are used to propose to the User and Server threads a transparent access mode to database pages These threads don't have to be aware of:
• page localisation on local disks or on disks managed by distant servers
• swap between disks and primary memory
• page locking

The database is composed of several volumes. If the WS directly accesses a volume, this volume is said to be local to the WS. If the volume is accessed via a server WorkSpace, it is called a distant volume.

In the case of a local volume, memory mapping is used so that the WS threads can directly map the volume (fig. 2a). The volume is composed of several segments, each one is a contiguous block of pages mapped in memory. A thread which must access a page maps the corresponding segment. The size of each segment is defined at its creation time. As the segment is the mapping unit for threads, this size will determine space consuming in the virtual memory of the WS.

The segment is mapped in a private mode by each thread: thus modifications remain private to each thread. At the commit time of the thread, modified pages are copied into the segment via a shared mapping of this one[1]. The previous image of each modified page is first copied in a before-images log. This log makes possible to undo the threads which would fail during their commit.

Page locking is executed by using the protection mechanism of the virtual memory. Initially, pages of a segment mapped by a thread are protected against read and write access. Afterwards, each new access to a page of this segment raises an exception which signals the protection violation (UNIX signal `SIGSEV`). A (read or write) lock is then asked to the lock manager. When the lock is granted (either immediately, or when the lock is relaxed by the owner thread(s)), the thread is then restarted from the faulty instruction.

[1] An optimization enables the Server Threads to use this shared mapping: they only modify pages at commit time.

In the case of a distant volume, a dialogue is established between the client WS and the server WS, for the importation of the needed pages throw the network (fig. 2b). Imported pages of a distant segment are thus copied in a local file which plays the role of local image. The WS threads have to map this image file for accessing the pages of the distant segment. This image is incomplete. When a page is not present in this image, a request for this page and its associated lock is sent to the server WS. User Threads are blocked up until the page return. Private mapping on the image file and locking detection are identical to those of a local segment. Image file mapping behaves as a local swap of the pages of distant volumes. The thread commit differs according to the operation mode of the WS to which it is attached:

• In the case of a passing WS, pages modified by each thread are copied into the image file and are also sent back to the server WS of the volume.

• In the case of a Retaining WS, modified pages are only copied into the image file after being locally logged. At commit time, the retaining WS sends back to the server pages modified by committed threads. As we shall see in the paragraph about concurrency control, this phase also corresponds to the release of write locks.

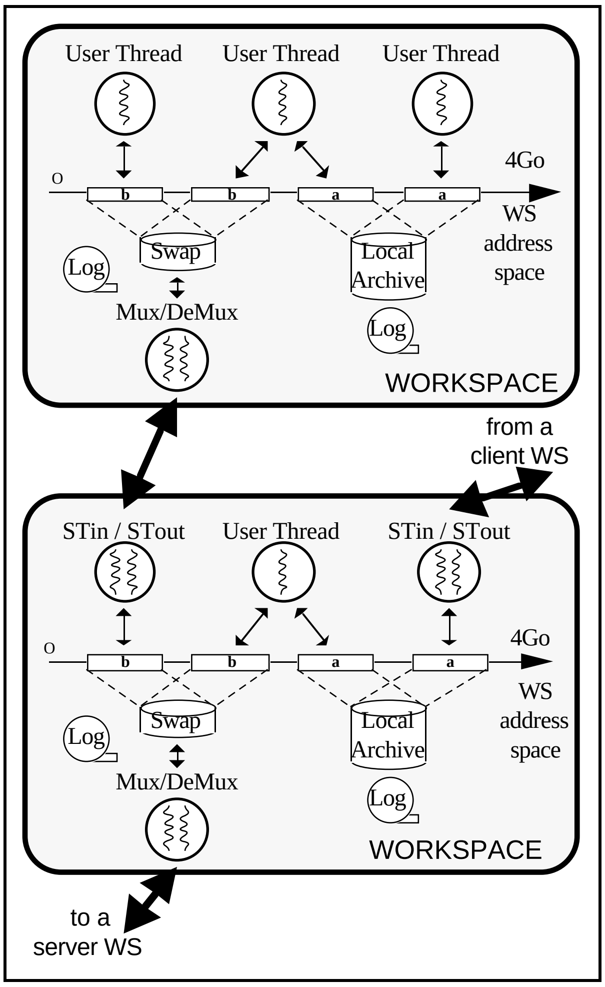

Figure.2-Internal operation of Workspaces: mapping local segments (a) and distant segments (b) by User and Server Threads.

Memory mapping theoretically enables to install up to 4 GBytes of objects in the virtual memory of a WS during its "lifetime". This limit is too low for longer service time Server WS.

Therefore, a reclaiming mechanism releases the mapping of some segments in order to allocate it to other ones, somewhat in the manner of a swap. It must be noticed that segments mapped by User Threads are not subject to swap.

4.3. The Object interface

The Page interface described at previous paragraph does not know the internal pages structure. It is completed by the Object interface which structures pages into object containers. The Object interface enables to create and destroy short and long objects, and executes their dereferenciation. At this level objects are not structured.

4.3.1. Short Objects, Long Objects

Short objects are not structured and receive instances of C++ classes defined by the Language interface. Short objects contain the identifier of the object class, and the memory pointer to the table of virtual methods of the C++ instance. This pointer is only valid for one WS, therefore it is updated (or "refreshed") each time it is stored in a new WS. A short object can also contain persistent references to other objects. The short object must be archived within a single page. The long object is dedicated to the storage of a single data type: byte strings, which can be multimedia data such as images and sounds.

At creation time, a persistent object may be placed "nearby" another existing object. In this case it is placed in the same page if it is possible. This object placement optimizes disk and network access when both objects are used at the same time.

4.3.2. Object identifier and dereferenciation

Object access is usually done through an object identifier, which is a unique reference through the whole database and which is not re-allocated [6]. The format of an object identifier differs according to the instanciation mode (persistent or temporary), and also according to the object type (short or long object). The identifier of a temporary object is a physical address. The identifier of a short persistent object is composed of a volume number (which is used to localize its server WS), a segment number within the volume, a page number within the segment, and an object number within the page. As a long object occupies an integer number of pages in a segment, the identifier of a long persistent object is composed of volume number, segment number, and the number of its first page within the segment.

Dereferenciation of an identifier is the translation of an object identifier into the memory address of the place where there is an image of this object. This operation is a critical one in persistent object managers. In WEA, this translation is done in two steps. The first step calculates the segment mapping address by linear hashing. Because hashing is done on segment number, the hash table consumes very little space. The second step is an indirection at page level and returns the relative position of the object within the page. This is done by indexing an indirection array within the page header. This step only implies a few instructions.

The identifier size is one of the configuration parameters of the system. The identifier can be coded on 32 bits or 64 bits. A 32 bit coding offers a maximum of 32 GBytes of short objects at a same time and 4 TBytes created in the whole database lifetime (for a database without long objects). The size of a database made of both kind of objects belong to this 32 GB - 4 TB range. The long objects database includes at most 4 TBytes of data at the same time (for a database without short objects). Coding on 64 bits offers a quasi-unlimited database size, but increases the size of persistent references within objects.

4.3.3. Object creation strategies

Temporary objects are allocated in a private, temporary area of the thread. At commit time, temporary objects are copied from the temporary area into already write-locked pages. If there is not enough free space, the thread applies for an interval of free pages in order to place the remaining objects. These new persistent objects receive a new identifier based on their new page address. References to these objects must be updated.

When there is no more room to create persistent objects in the thread or WorkSpace pages, two strategies are possible. First it is possible to create new pages if needed during transaction. A second strategy is to create the objects in the temporary area, and wait for the commit to place these objects in other pages obtained and locked further. This strategy returns a better page occupation ratio, but implies a copy of objects within memory.

4.4. The Language interface: objects typing

While the Objects interface manipulates non structured objects, the role of the Language interface is to manage a metabase of the C++ classes used by Workspaces, and to instanciate these classes in the case of short objects.

4.4.1. Tools

This interface uses a parser and a C++ compiler. The parser extracts the list of the C++ classes and adds them to the database. For each of these classes, it also generates some methods code. The operator `->()` overload the dereferenciation operation, and some methods "refresh" the pointer towards the virtual methods table of the class. Each instance includes a header, with the corresponding absolute class number. The C++ compiler is any compiler of the marketplace

(ATT, GNU,…). It generates the code of applications from the source code delivered by the developer, and from the methods produced by the parser. Standard development tools such browser or debugger remain usable.

4.4.2. Transaction Example

We consider the following database schema (fig.3) and two transactions (fig.4a and fig.4b):

Persistent classes definition is identical to C++ semantic. A class may contain members of fundamental types (lines s3-s7) or pointers to instances of persistent classes. A member may be a standard structured type such as `union`, `class`, `array` (s3-s4) or a generic collection defined in the WEA library (s5). The `colldef` keyword defines new objects collections. Class definition also includes declarations of virtual or ordinary methods. These methods can be "inlined" or defined in the user library.

```
s1. colldef List(Person) PersonList;

s2. class Person { // private:
s3.    char        name[20];
s4.    p_Person    parents[2];
s5.    PersonList  children;
s6. public:
s7.    int age; // public member
s8.    inline Person() { age = 0;}
s9.    Person( char*    name,
               p_Person father=NULL,
               p_Person mother=NULL);
s10.};
```

fig.3. the "Bible" Database Schema

Database access begins with a connection of the WS to the database (a1). Transactions may be then instanciated on this WorkSpace(a2). Persistent objects are created as C++ objects with the `new()` operator (a3). An object may be placed nearby another one at creation time (a4). It may be created as temporary (b5) and then made persistent before transaction end (b7). An object may be named by an external name (a5-a6) and thus it can be retrieved during another transaction (b3-b4). Object members are handled in a classical way by the `->()` operator in functions (b6) or in methods (s8). C++ Member Access Control is respected (s2-s6). Member updates implicitly locks the object page (b6). A transaction ends by a commit or an abort (a7).

```
a1. WEA::connect("Bible");
a2. W_Transaction *trans = WEA::newTrans();

a3. p_Person eve = new Person("Eve");
a4. p_Person adam= new(eve) Person("Adam");

a5. adam->setName("FirstMan");
a6. eve->setName("FirstWoman");

a7. trans->commit(); // or trans->abort()
a8. WEA::disconnect();
```

fig.4a. initialize the "tree of life".

```
b1. WEA::connect("Bible");
b2. W_Transaction *trans = WEA::newTrans();

b3. p_Person fman("FirstMan");
b4. p_Person fwoman("FirstWoman");

b5. p_Person abel=
      new(TEMP) Person("Abel",fman,fwoman);

b6. abel->parent[1]->age += 1;
b7. abel->persistent();

b8 .trans->commit(); WEA::disconnect();
```

fig.4b. add an new Person in the "tree of life".

5. DBMS FUNCTIONALITIES

5.1. Concurrency control: hierarchical Callback Locking

Wang and Rowe compare several concurrency control methods for client-server database architectures [13]. The results of their simulations underscore the advantages of the Callback Locking method (CL). CL is based on two-phase locking (2PL), but it keeps on the client a cache of read locks obtained by its transactions. Write locking, on the opposite, implies to invalidate the read lock in each client cache. This read lock is released by the client WorkSpace when it is asked for and when all reading transactions have committed. The server WorkSpace is used to centralise and broadcast the lock queries. This method is specially adapted to situations where transactions of a same client reference the same objects. This often should be the case in applications which use Workspaces.

We have adapted Callback Locking to the hierarchical organization of the WorkSpace architecture. Each nesting level becomes a lock server for the upper level. However, the WS can chose to inhibit the read lock cache and to locally operate in a way similar to classical 2PL.

The WorkSpace operation mode also influences the release of write locks. Passing WorkSpace releases write locks after each thread commit, while the Retaining WorkSpace keep them until its own commit.

The lock granule is the page. This grain enables detection of locks queries by the operating system virtual memory mechanisms, which is not possible with Object grain.

The WorkSpace does not know precisely the status of locks set by its clients threads. Therefore it is not possible to implement a dependency graph between threads to detect deadlocks. Deadlocks are detected by a time-out, which raises an exception in the thread.

5.2. Reliability and restart.

Restart is implemented in a classical way with a before-image log. In a Retaining WorkSpace, logging is also used to undo the threads which abort modifications done on distant volume. In a future version of WEA, these before-images may also be used to extract historical versions of data.

5.3. Security and access rights

WEA uses the access rights mechanisms and identification mechanisms of Unix. The WorkSpace possesses the user identifier (UID) and the group identifier (GID) of the user who starts it. In a volume, all objects have the same access rights (owner, group, other). Using this control level do not introduce hole in the system security.

6. CONCLUSION

WEA is a generic brick to design distributed applications corresponding to different models (classical client-server, application server, etc…). It offers a transparent access to a large virtual memory of objects, which is distributed on a network of workstations. Its C++ interface enables it to benefit from the power of this language. Internal design of Workspaces uses advanced techniques proposed by the new operating systems. WEA is at the cross-road between system and language, which enables it to benefit of technological advances, while remaining independent of implementations.

Design of WEA begun in October 1992 and a first prototype will be completed in December 1993. Future works may extend the WorkSpace to support cooperative work and versions. This extension implies a specialisation of the data model and of the transactional model. We also intend to use WEA as the software environment for a powerful, object-oriented, associative board [14].

REFERENCES

1. M.J. Carey, D.J. DeWitt, J.E. Richardson, E.J. Shekita, "Object and File Management in the Exodus Extensible Database System", VLDB'86.
2. O. Gruber and L. Amsaleg and L. Daynes and P. Valduriez, "EOS, An Environment for Object-Based Systems", Proc. of the 25th Hawaii International Conference on System Sciences", January 1992, Vol 1.
3. O. Deux et al, "The Story of O2", IEEE transactions on knowledge and data engineering vol.2 n°1 1990
4. ATT Bell Laboratory, "ODE 2.0 User's Manual", 1993
5. C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", Communication of the ACM October 1991, Vol. 34, No. 10
6. S. Khoshafian and G. Copeland, "Object Identity", MCC Research Report, 1986.
7. IEEE, "POSIX 1003.4a Draft 6 - Threads Extension for Portable Operating Systems", Feb 1992.
8. SunSoft, SunOS 5.2 Guide to MultiThread Programming, Answer Book, March 1993
9. E. Abécassis, Private Communication, 1993
10. M.A. Ellis, B. Stroustrup, "The Annoted C++ Reference Manual", Addison-Wesley, 1991
11. J.F.Garza, W Kim, "Transaction Management in Object-Oriented Database System", ACM SIGMOD 1988.
12. SunSoft, SunOS 5.2 Memory Management, Answer Book, March 1993.
13. Y. Wang, L.A. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture", ACM SIGMOD 1991
14. D. Archambaud, P. Faudemay, A. Greiner, "RAPID-2, An Object Oriented Associative Memory Applicable to Genome Data Processing", 27th Hawaiin International Conference on System Sciences, January 1994.