YOODA : Handling Distribution through OODBMS

E. Abécassis

APIC Systems
25 rue de Stalingrad
94742 Arcueil Cedex France
ea@apic.fr

Abstract

YOODA is an environment for handling distribution. It is based on an object oriented database extented toward distribution. The main extension is a model, the Workspace Model, which allows data and control distribution. The low-level ofYOODA is an encapsulation of operating system services such as multi-threading or memory mappings. The high level proposes a distributed database, a C++ extension for persistentce and a nested transactions protocol. This leads to the definition of the YOODA process which is an unification of client and server processes in a classical database architecture. The Workspace Model is based on YOODA process to allow distribution.

1 Introduction

Client/Server architecture is one of the main evolutions in the computer field. Very high bandwidth networks and powerful workstations at low prices completely change the design of an application. New software architectures are based on open systems. Applications are made up of components, each of them focusing on a specific goal. Integrators build their applications by using appropriate components for each task. This vision of the domain requires new ways of handling distribution over a network.

In recent years, several domains have explored distribution handling. These approaches can be roughly divided into two groups; those that design a new operating system [DAS88], [SHA89] and those that use standard mechanisms to model distributed computing systems. On one hand, introducing a new operating system is a very long-term task that may take years even for the biggest companies such as IBM or DEC. Unix has spent twenty years in becoming an industrial standard. On the other hand, using standard mechanisms like multi-threading or network communication is only the first step to enable distribution. We must also provide a way to control distribution.

Providing distribution on top of a classical operating system is done by using dedicated tools; enhanced communication protocols such as CORBA specification [COR91] and multi-threading features. Handling of control and data distribution is commonly achieved through a transaction mechanism. This notion has been ‘stolen’ from the database field and made a mechanism of handling distribution. The major drawback of this ‘theft’ is the oversight of data sharing and recovery techniques that are needed to allow transactional computing (eg: abort a transaction, synchronise multiple threads, etc...). In this approach, there is no unified vision of the different techniques involved.

A great deal of research in OODBMS is to merge it into the operating system [SHE90][OBJ92]. In this way, it becomes a basic tool for building components which handle and share persistent (or transient) objects. Moreover, enhanced transaction management allows the use of OODBMS to build complex distributed architectures. An OODBMS transparently ensures data distribution, data sharing, data recovery and transaction management. Consequently, through evolution of the distribution model, OODBMS can be a good candidate to unify mechanisms involved in handling distribution with a smooth evolution and not a revolution.

In this paper, we describe YOODA as an OODBMS for building distributed applications. The system provides persistent and concurrent language based on C++. Communication takes place through a CORBA like interface. By using an analogy between threads and transactions, YOODA provides an easy way to handle synchronisation and recovery in a multi-threading environment. Moreover, YOODA focuses on performance giving a single view of transient and persistent objects.

The key feature of YOODA is the encapsulation of low-level system functionalties to provide a consistent model for distributed applications. Memory mapping, network communication, multi-threading are integrated in a high level environment. On top of this, YOODA proposes a simple model as a framework in order to build distributed applications. This model is called the Workspace Model.

The Workspace Model has been worked out by a team distributed in MASI laboratory, UTC laboratory and APIC systems. Several implementations of this model are currently being studied. WEA [DON94] is one of them. It focuses on dedicated architectures for cooperative work. YOODA focuses on an implementation of the Workspace Model usable in industrial projects. It is currently in use in several research teams and in two large industrial projects. The current version of YOODA is available through FTP at ftp.ibp.fr.

This paper is organized as follows: section 2 discusses related work on OODBMS and transaction management. Section 3 presents the Workspace Model. Section 4 describes the implementation of YOODA. Section 5 discusses the performance results of OO7 benchmark. Section 6 contains some conclusions and proposals for future work.

2 Related Work

YOODA is defined as an evolution of a classical OODBMS to handle distribution. For this purpose, it must provide object oriented database and distribution functionalities. In this section, we present some systems which provide some of the features that YOODA may unify. First, we describe the classical OODBMS (EXODUS[CAR90], Objectivity, Ontos[ONT91], ObjectStore[OBJ92]) which appeared in the benchmark results at section 6. Then we discuss the Shore system. This project addresses similar problems to ours in a different manner. We close this section by a short presentation of enhanced transactions managers.

2.1 Classical OODBMS

Exodus

At the end of the eighties, several projects studied how to define a basic kernel for DBMS [STO90][LIN87][BAT88][CAR90]. EXODUS is one of them. Basically, EXODUS is made up of two components; a storage system (ESM) and Persistent Virtual Machine (EPVM). The architecture is based on a classic client/server model. The storage manager is a page server. It is in charge of page access, concurrency control and recovery. Locking is provided at page level with a two phase locking scheme (2PL). Recovery is based on logging the changed portions of object. At the user level, EXODUS provides E language [RIC90] as an extension of C++. E language inserts calls to the Persistent Virtual Machine (EPVM). The EPVM is in charge of local cache management and communications with the server. The cache supports a comprehensive pointer swizzling scheme [MOS90][WHI92] that swizzles inter-object pointer references i.e. converts pointers from object identifiers (OID) to direct memory pointers. The main purpose of EXODUS was to define a kernel to build dedicated DBMS. ESM and EPVM are the basic blocks to do that. Enhanced transaction management and distribution are not taken into account.

Objectivity

Objectivity is a commercial product available as DEC Object/DB. It is based on a file server architecture. There is no server process for handling data. Clients access pages via NFS. A separate process is used as lock server. Locking is acheived through a 2PL scheme. Recovery is implemented via shadows. Unlike EXODUS, a library based approach is used to provide persistence to C++ objects. Persistent objects are defined by inheritance from a persistent root class. The transaction management is a classical ACID transaction scheme (Atomic, Consistent, Isolation, Durability).

Ontos

Ontos uses a client-server architecture like Exodus. The server process manages page access, concurrency control and recovery. Locking is also achieved through 2PL. Recovery is via redo logging. Like Objectivity, persistent objects are defined by inheritance from a persistent root. Cache management at the client level is achieved through virtual memory. Ontos keeps objects in virtual memory under the control of the client cache. This approach limits the amount of objects a client can access in a single transaction. Transaction management is a classical scheme with nested extension. It is not nested transactions in the sense of Moss [LYN86] but a heap of transactions. A transaction can start only one sub-transaction at a time.

ObjectStore

ObjectStore is a commercial product which focuses on performance. It is based on a client-server architecture with a page server. The key feature of ObjectStore is the use of memory mapping techniques and pointer swizzling. Memory mapping means that the database itself is mapped into the virtual address space of the client, allowing persistent data to be accessed in the same manner as non-persistent data. By this way, it can achieve the same level of performance as C++ code. ObjectStore provides a multi-server architecture. A client can open several databases, each of them is handled by a different server. The transaction management is based on a 2PL scheme with read and write locks. Like Ontos, it supports transactions heaps. Cooperation is possible through the use of version and configuration management schemes, with check-in and check-out facilities. But ObjectStore does not provide any communication or notification mechanism.

2.2 Distributed OODBMS

SHORE [CAR94] (Scalable Heterogeneous Object REpository) from Wisconsin-Madison university proposes a new architecture for object manager. SHORE is a distributed object server. It proposes a data language definition to modelise objects. Classical constructors and basic types are proposed. The architecture is a peer-to-peer scheme. Each node on the network has a SHORE server. Clients always ask for objects from the local SHORE server. This one looks for the object in local cache. If it is not present, it retrieves the object from the SHORE server which archives it. The commit is achieved through a Two Phase Commit Protocol (2PC). The coordinator of the commit is the local server. SHORE has a multiple adaptative grain locking scheme. It is based on a classical 2PL with data replication.

SHORE provides database functionalities at low level. It is merged in the operating system and can be used as a high level file manager. Data stored is not only a meaningless sequence of bytes but structured objects. Distribution is provided transparently through the network of SHORE servers.

2.3 Enhanced Transaction Processing Prototypes

CAMELOT [SPE89]

CAMELOT is a distributed transaction facility research prototype developed at Carnegie Mellon University. It is intended to support wide-spread use of transaction processing techniques. It executes on a variety of uni and multi-processors on top of Mach operating system. Most of CAMELOT functionalities are implemented by a collection of Mach processes, which run on every node in a distributed system. The main processes are:

- disk manager: it allocates and desallocates recoverable storage.
- Communication manager: it forwards internode Mach message and provides name and clock services.
- Transaction manager: it coordinates the initiation, commit, abort of local or distributed transactions. It fully supports real nested transactions [LYN86].

The CAMELOT experience has shown that an important work is required to provide a transactional support. All the multi-threading, synchronisation and recovery techniques are difficult to implement. CAMELOT starts from the transaction concept and has implemented everything needed for that purpose. The work presented in this paper starts at the opposite end. It uses communication and storage techniques of OODBMS to build transaction management adapted to distribution.

QuickSilver [SCH91]

QuickSilver implements a transaction processing system at the operating system level. QuickSilver is an experimental distributed OS that was developped at the IBM Almaden Research Center. It was designed to make it easy to write sophisticated distributed programs. QuickSilver extends the notion of transaction to serve as the method used for all resource management in the system. Every program runs in the context of transactions. QuickSilver environment is composed of ;

- Transactional IPC: communications must be done on behalf on a transaction.
- Transaction Manager: it handles initiation and termination of distributed transactions. The model used is a traditional 2PC protocol.
- Log Manager: the transaction manager uses the log to recoverably record the state transitions of transactions.

QuickSilver does not support nested transactions and hence is not really adapted to distributed transactions. Furthermore, it does not provide long transactions.

3 Overview of the YOODA's Workspace Model

3.1 Requirement

YOODA attempts to provide an integrated environment for programming distributed applications. As mentioned earlier, it must be able to manage data distribution, data sharing, data recovery and enhanced transaction management. YOODA also provides a model which is based on those functionalities to build distributed applications.

The first step is to offer a complete, low-level OODBMS. Low-level means that YOODA does not address query language or data-model evolutions. It focuses only on a persistent language like E/Exodus. A major constraint we put on this goal is to develop a portable and usable OODBMS with only the fewest assumptions on new operating system functionalities. YOODA must be able to run on top of classical Unix kernels. The second step is to provide distribution tools. The goal of YOODA is to unify the different techniques used in building distributed applications into a consistent model, the Workspace Model.

Requirements on the Workspace Model are defined by the analysis of what could be an extension of an OODBMS towards distribution. Our study focusses on data distribution and control distribution.

3.1.1 Multi-server

A YOODA database is made up of volumes. A volume is a container of objects. It is an independent unit of storage. Independent means that its internal structure is not specific to the process which manages it. When a process manages a volume, it is responsible for objects accesses, concurrency control and recovery. In a classical client/server architecture [DEW90], every volume of the database is managed by a single server. In the Workspace Model, volumes are distributed between several servers. Each server can manage several volumes. The topology of the database is statically defined in a file by naming every server and every volume.

Each YOODA process contains a component which acts as a virtual volume server (VVS). The VVS uses topology file to establish connection with the physical servers. When a client requests an object or a lock, its VVS finds the physical server which is responsible and sends a demand to it. The VVS hides distribution of the database from the upper layers.

3.1.2 Private database

A possible extension of the VVS is to allow management of local volumes. VVS hides distribution through an indirection table. This table can lead to remote physical servers or to local volumes. In this way, every YOODA process can have several local volumes. These volumes act as a private database.

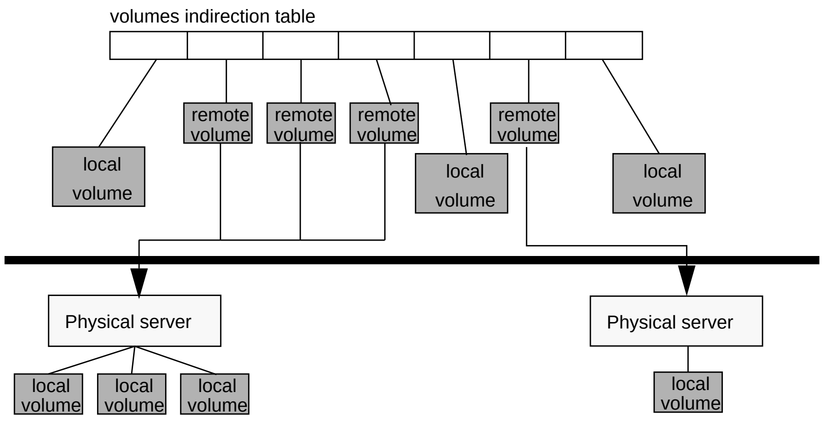

Figure 1. Remote and local volumes management through VVS

3.1.3 Parallelism

One approach to achieve parallelism is to use multi-threading. Recent Unix kernels (Solaris, OSF, NextStep) provide transparent parallelisation of multiple threads over available processors. The main difficulty of this parallelism is the synchronisation of the threads. Multiple threads share the same address space. Programmers must take care of critical resources. To handle this problem correctly, multi-threading layers provide a set of synchronisation tools (monitor, semaphore...), but it is still a complicated problem. YOODA simplifies multi-threading management by using nested transaction models. For that purpose, we introduce two notions:

- Control Transaction. This is the main transaction of a YOODA process. At the beginning of a YOODA process, only control transaction is active. Then it can start threads or do local treatments.
- Activity. An activity is a sequence of transactions which are executed into a thread. Each transaction of an activity is defined as a subtransaction of the control transaction (in the sense of Moss Nested Transaction).

Through the analogy between thread and activity, we have a simple technique to handle parallelism. Activity can be defined as a thread with transactional properties. Therefore, synchronisation is realised by OODBMS and parallelism by OS. The programmers work becomes easy.

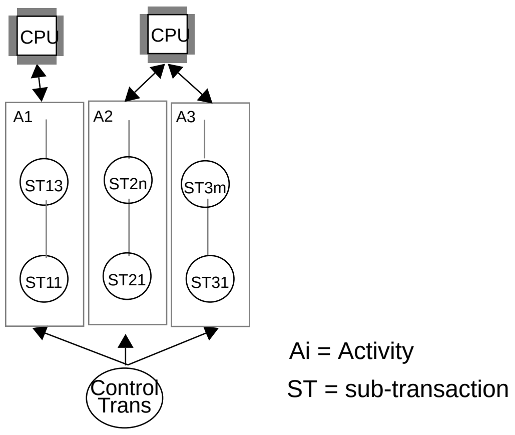

Figure 2. Parallelism using threads

3.2 YOODA Process : Result of Unification

Through the unification of the three previous concepts, we no longer need the server notion. A server is specialised YOODA process which contains one activity for each possible client. Sub-transactions of the activity are proxies of the transaction executed in the remote client. Moreover, every process can use some of its activity to serve remote subtransactions through the nested transaction protocol provided in YOODA.

3.2.1 Definition

A YOODA process is made up of:

- Virtual Volumes Server. This componant enables a simple view of the database. Distribution of the volumes is hiden from the upper layer. A volume can be local. Then it is managed directly by the local VVS. A volume can be remote. Then it is managed by a remote YOODA process.
- Control Transaction. This transaction allows simple computing on the database. It can also launch activities.
- Activities. Each activity is a sequence of sub-transactions of the Control Transaction.
- Sub-transactions.Every sub-transaction belongs to an activity. A sub-transaction can do local computing or only be a proxy of a remote transaction. In such case, the Control Transaction of the remote process is a subtransaction of the local Control Transaction.

A Client process is a YOODA process which is connected to several remote servers to have a version of the database.

A Server Process is a process which manages one or more volumes of the database. A Server Process can refuse to serve remote Client. In such case, volumes managed are private and can only be used for local treatments.

A Workspace Server Process is a process which provides activities to handle remote processes as subtransactions.

Local view of the database is called a Workspace. It is made up of remote objects stored in the process's cache and of local objects stored in the local volumes.

3.3 The Workspace Model Definition

Considering a database made up of volumes V1, V2,..., Vn

Considering Server Process S_1, S_2, ..., S_m

We distribute volumes over the servers by following topology defined for that database.

We call T_{RAC} the transaction which corresponds to the stable version of the database.

A YOODA Process can be connected to all of the Server Processes or to only one Workspace Server process.

Every YOODA process connected to all the servers is considered to be a sub-transaction of T_{RAC} in the sense of Moss Nested Transactions.

Every YOODA process connected to a Workspace server is considered to be a subtransaction of the control transaction of its server.

3.4 Possible Use of the Workspace Model

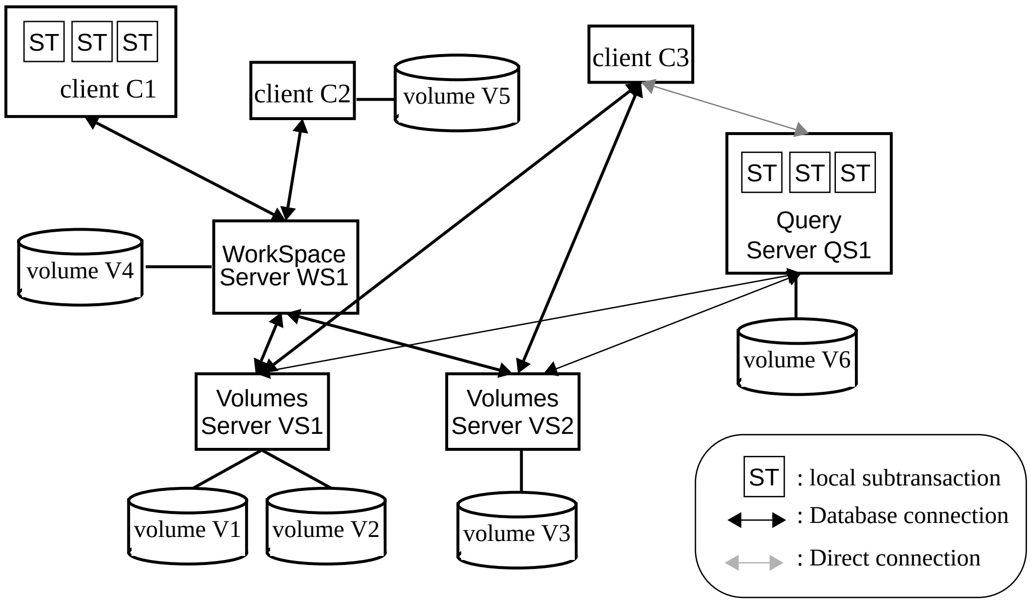

Figure 3. An example of a YOODA architecture

Figure 4 illustrates a possible architecture with the Workspace Model. At the bottom of this archictecture, two servers VS1 and VS2 manage volumes V1, V2 and V3. These servers are just like classical servers in a multi-server OODBMS. They use activities to serve remote clients in the context of transactions.

The Query server is a YOODA process which manages a private volume (V6). This volume is used to store indexes on data contained in volumes V1, V2 and V3. It implements a query processing service through activities. A client can be connected to one of these activities, send queries to it and get the results. Note that connections between a client and a query activity is not done through nested transaction mechanisms. It is directly connected through the CORBA communication protocol provided in YOODA.

Client C3 is a classical OODBMS client. It is connected to volumes V1, V2 and V3. It cannot access directly to volume V6 but can send queries to the Query server.

The Workspace Server WS1 is connected to V1, V2 and V3 through the servers SV1 and SV2. It is directly connected to volume V4. It manages a version of the database made up of a stable version of V4 and a version of V1, V2 and V3, modified by its commited subtransactions. When WS1 commits, it archives its own version of V1, V2 and V3 back to the servers. It uses activities to serve its own version of the database to remote clients. Workspace Server implements the basic mechanisms of groupware and cooperative works.

Client C1 and C2 share a private version of the database made up of volumes V1, V2, V3 and V4. Every commit of C1 or C2 is only visible to them. The other clients will see modifications only when WS1 commit. C2 manages V5 as a private volume.

4 YOODA's System Architecture

Architecture of YOODA is divided into four parts. At the lowest level, we have the Virtual Volumes Server. It manages the database volumes (remote and local) and YOODA's cache to store remote objects. The object server provides a view of the volumes as a collection of objects. The transaction manager is responsible of the concurrency control and commit/abort of local and remote subtransactions. Lastly, YOODA/C++ provides a C++ interface to handle persistent and transient objects.

On top of this architecture, the user library specialises the YOODA process as a volumes server, a Workspace server or a simple client.

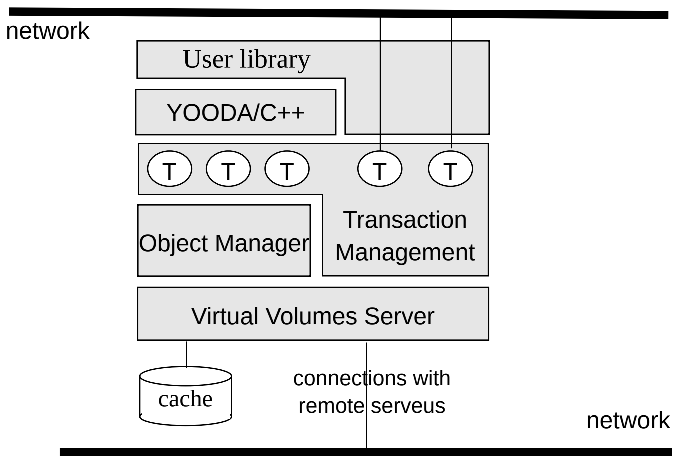

Figure 4. YOODA architecture

4.1 Virtual Volumes Server

A YOODA Database is based on Unix files. Each file, called a volume, is divided into pages. The page is the distribution unit which allows a simple implementation of the Workspace Model. A page is:

- a storage unit. Every object is stored in pages
- a virtual memory unit. Local cache is made up of pages. Therefore, we can use memory mapping techniques to manage virtual memory.
- a transfer unit. Communication flow is better if transfer units are large.
- a locking unit. Setting a lock implies communication. To avoid excessive communication, we choose to handle locking grain at the page level.

Each volume is viewed as a sequence of pages. VVS proposes a transactional interface for these pages. Higher layers can read a page, lock a page, start a commit, write a page, allocate a page, end commit, abort...

To handle distribution, the VVS uses an indirection table. At the beginning of the session, VVS reads the topology file to store network address of every server. The local volume entries point to the physical local server. A VVS recognizes itself by its name. A special name is used when we want to have every volume managed as local volumes, ignoring the topology file.

The local cache is a set of pages. When a page is asked to be read, the VVS allocates a page in the cache. The original page is copied into the cache and a virtual memory address space is allocated through the memory mapping. The memory address is available until the end of the current transaction. Each subtransaction has different mapping for the cached pages but all of them reference the same physical copy of the page until one of the subtransaction attempt to modify it.

To lock a page, we use the protection mechanism available through memory mapping. Initially, every page is protected against write. When a higher level attempts to write a page, the VVS receives a Unix signal. The lock is requested from the responsible server and if it is granted, page protection is modified to allowing write.

4.2 Object manager

At this level of the architecture, objects have no semantic information. An object is a byte sequence of an arbitrary size identified by an OID [KOS86]. The object manager allows to get a memory pointer on an object from its OID. Object identifier is a physical OID. It is composed of volume number, page number in the volume, object number in the page. The choice of a physical identifier is justified by three reasons. First, for performance goal, the use of a physical identifier is better. Second, the property of independence of the volume could only be achieved through physical OID. Lastly, YOODA objects are essentially used through the YOODA/C++ language. In such language, the objects rarely move, except for migration purpose. Consequently, physical OID seems to be a good choice. In-memory decoding of OIDs is done by an efficient hashing structure. Performance results justifie our choice not to use swizzling. Swizzling is only used for local variables.

Object allocation is done through clusters. A cluster is a set of pages of a volume. A volume can hold an arbitrary number of clusters. A cluster can hold an arbitrary number of pages. When an allocation is requested by higher levels, the object manager finds the specified cluster where the allocation must be done. Each cluster maintains an allocation structure made up of loaded pages. Each time a page is loaded, it is inserted in its cluster allocation structure. To make the allocation, the cluster looks for a page with an empty slot of the right size. Search is first done through write-locked pages. Allocation never implies waiting for locks. If a lock cannot be granted, search continue on other pages. This method allows simple distribution because every process first uses its local resources to make an allocation. Remote requests are only sent to allocate new pages in a volume when allocation cannot be achieve with local resources.

The Object Manager handles objects of arbitrary size. Objects smaller than one page are always allocated inside a single page. Objects larger than one page are stored on an integer number of pages. When a long object is referenced, only the first page is loaded but virtual memory space is allocated for all the object and every pages but the first one are protected against read and write. When upper layers attempt to read unloaded pages, a Unix signal is sent to the VVS which loads the page from the volume. This method allows incremental and transparent loading of long objects. Memory pointers to a virtual contiguous space can be sent to the upper layer (which is valid only during the current transaction).

4.3 Transaction Manager

The transaction management can be broken down into five components; Control Transaction, Remote Transaction Client, Local Transaction Client, Remote Transaction Server and Local Transaction Server. The main component is the Control Transaction (CT). It manages concurrency control, recovery and two phase commit at the process level. A CT is a distributed transaction. It is divided into one Local Transaction Client (LTC) and several Remote Transaction Clients (RTC). The LTC is responsible for the local version of the database. The Local version is made up of stable version of the local volumes. Remote Transaction Servers (RTS) are subtransactions of the CT of the servers. RTC are proxies of the RTS based on the process's servers. The last component is the Local Transaction Server (LTS). It is a local subtransaction of the process's CT. It is used for the local activity facilities based on multi-threading.

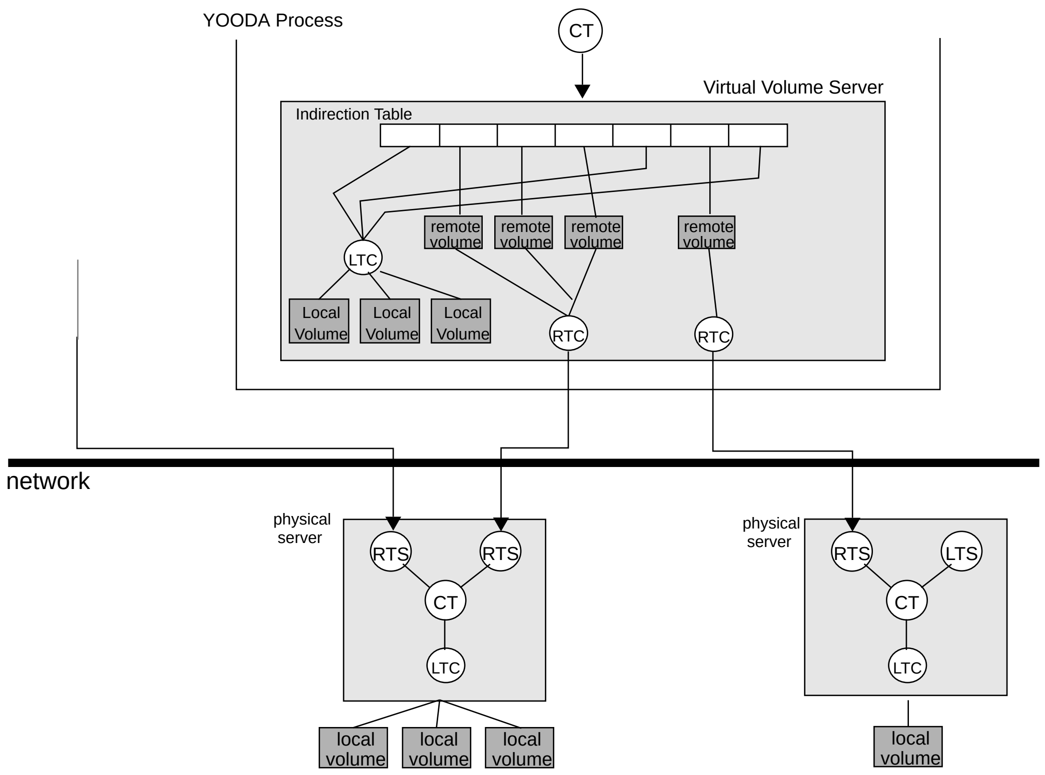

Figure 5. Transaction architecture

Concurrency control is done with a 2PL protocol. During the transaction, locks are requested on the pages. At the end of the transaction, every modified page is archived by the servers in their workspace and locks are released. Locks are granted at the page level. A transaction can ask a read lock or a write lock on a page.

Classical lock protocol implies a high level of contention. Indeed, a write lock on a page prevents other transactions to read this page. To reduce contention, the transaction management allows a special transaction mode where read locks on a page can be granted even if it is already locked in write mode by another transaction. To provide this functionality, transaction management guarantees that the transaction's view of the database is a snapshot of the stable version of the database at the beginning of the transaction. Write ahead logs are kept to register multiple database versions needed by running transactions in such mode. This technique is called Deltalog.

Commit of a subtransaction (LTS or RTS) is archived in the cache for remote volumes and directly in the volumes for local volumes. Workspace Commit is managed by the CT with 2PC protocol. The first step is to signal commit start to every LTC/RTC which contains modified pages. Then every modified page is sent to the servers. During all this phase, a log allows an abort of the transaction. When the servers have finished to store pages, the second phase can start. CT signals finish the commit to the servers.

4.4 C++ Interface

At the Object Manager level, objects are only sequence of bytes. YOODA provides an extension of C++ to add persistent property on C++ objects. Then, the objects used by developpers are C++ objects stored in a YOODA database. Basically, we identifie functionalities we must have to transparently handle C++ persistent object:

- Allocation
- Desallocation
- Data access
- Method call.

Allocation and desallocation are services provided at the Object Manager level. Simple redefinition of the new and delete operators for each persistent class allows the creation and deletion of persistent object. Data access and method call can be achieved through pseudo-pointers which encapsulate OID. This is done by overloading the class member access '->' . Therefore, the C++ can normally access to the data members or function members.

Programming with a database implies basic tools such as collections, index, etc... YOODA provides efficient lists, dictionary and string to C++ programmers. There are provided as C++ templates.

To use C++ interface, the programmer just have to declare his classes into the YOODA'schema utility. This tool creates a modified version of the classes, directly usable as the originals. The tool creates also pseudo-pointers for each class. The programmer must use this pseudo-pointers instead of normal pointers to reference persistent objects.

5 Benchmark

5.1 OO7 Benchmark Description [CAR93]

The OO7 benchmark has been worked out by a research team in the University of Wisconsin. It is a good compromise between complexity and testing capabilities. it covers a large range of OODBMS functionalities. It has quickly become the most famous benchmark for OODBMS.

5.1.1 Database description

The OO7 schema attempts to model a classical CAD or CASE applications.

The key element of the schema is the composite part. A composite part is made up of a couple of attributes, a document which is a large text and a graph of atomic parts. The atomic parts are connected through connection objects. Number of connections for an atomic part is a benchmark's parameter.

The composite part are grouped into a hierarchy of assemblies. At the lowest level, the base assemblies are made up of composite parts. The assemblies are grouped into complex assemblies along the tree. One tree of assemblies composes a module. A module contains a tree of assemblies and a manual which is large text.

The benchmark defines three different databases, the small one, the medium one and the large one. Another criterion is the number of connection in the graphs of atomic parts.

	Small	**Medium**	**Large**
Atomics per composite	20	200	200
Connections per atomic	3, 6, 9	3, 6, 9	3, 6, 9
Document size	20K	200K	200K
Manual size	100K	1M	1M
Composites per module	500	500	500
Assembly levels	7	7	7
Composite per assembly	3	3	3
Number of modules	1	1	10

5.2 Results

Proposed hardware to realize the benchmark is an isolated piece of Ethernet with only two machines. The first one is a SUN sparcstation ELC with 24 Mbytes. It runs the client. The other one is a SUN sparcstation IPX with 48 Mbytes to run the server.

Hardware used in the results presented here is slightly different. We run the benchmark on an Ethernet with about 30 machines. One of theim is a SUN sparcstation IPX with 24 Mbytes. It is used to run the client. Another one is a SUN sparcstation IPX with 32 MBytes. It is used to run the server. The performance ratio between IPX and ELC is approximatively 1.25. This small difference is not signifant because most of the benchmark uses virtual memory and then, I/O.

The code used for the benchmark is a modified version of Exodus code. It has been picked up from the ftp site of the Wisconsin University and modified to be in the YOODA syntax. Queries have been handcoded because YOODA doesn't have any query language.

TABLE 1 Database sizes

	Exodus	**Ontos**	**Objectivity**	**Objectstore**	**Yooda**
S3	11,5	4,2	5,7	4,4	7,1
S9	15,9	4,9	10,1	7,7	13,2
M3	103,8	51,7	54,6	37,5	55,3
M6	125,6	122,3	74,9	55,4	71,8

5.3 Traversal

TABLE 2 Traversal t1

	Exodus	**Ontos**	**Objectivity**	**Objectstore**	**Yooda**
S3 cold	34,8	28,9	38,5	22,7	19,9
S3 hot	10,6	8,1	17,9	6,2	5,1
S9 cold	50,6	63,7	66,5	45,5	36
S9 hot	15,1	16,2	36,8	12,4	8,9

	Exodus	Ontos	Objectivity	Objectstore	Yooda
M3	734,5	1064,6	548,7	372,5	322,3
M6	965,4	2516,4	1269,4	808,9	1132

Traversal t1 picks a module and traverses its assemblies. For each base assembly visited, visit each of its composite parts. For each composite part, do a depth-first-search on its graph of atomic parts. In S3 and S9, most of the database stay in physical memory. The good cold results for YOODA are the effect of the simple page management. M6 result is bad because of a problem in cache management. In t1, the database is loaded four times in the client cache. This one is too small for M6 database. Then, t1 consists to read four times the complete database from the server. This explains bad results for M6. We can observe the results in hot traversals for S3 and S9. There are similar to ObjectStore although this one uses swizzling techniques. This is because swizzling is much better if code only does dereferencing. In general purpose code, the gain is not always noticeable.

TABLE 3 Traversal t1 in multiple transactions

	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	34,8	30,0	40,2	27,9	19,4
S3 hot	13,8	21,2	22,6	7,0	7,9
S9 cold	50,0	73,2	na	51,4	36,3
S9 hot	22,4	47,8	na	14,2	15,0

This is the same operation but multiple traversals are done in different transactions. This tests inter-transaction cache. In YOODA and ObjectStore, the client cache keeps locks and objects. This is why the hot results are very closed to those in table 2. Exodus only caches objects. Ontos and Objectivity don't cache anything between transactions.

TABLE 4 Traversal t2a, t2b, T2c

t2a	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	39,3	39,2	61,0	35,7	37,9
S9 cold	59,9	83,2	96,9	67,0	67,7
M3	968,5	910,1	1356,0	567,2	513
M6	1227,9	2277,4	2199,4	1050,9	888,3

t2b	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	40,5	39,8	60,8	35,9	41,7
S9 cold	61,0	84,9	94,8	66,5	59,9
M3	963,3	901,7	1329,4	519,6	527,3
M6	1212,0	2243,4	2107,4	1052,3	894,5

t2c	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	36,3	38,2	64,3	35,1	35,3

t2c	Exodus	Ontos	Objectivity	Objectstore	Yooda
S9 cold	51,5	86,6	97,2	66	53
M3	759,3	839,2	695,9	461,3	328
M6	1000,8	2101,2	1468,8	961,7	581

The previous tables show the update functionalities. They use the same traversal as t1 but for each composite part update; the root part of graph (t2a), every atomic part (t2b), every atomic part four times (t2c). Simplicity of page level commit allows good performance. Note that M6 results are much better thant in t1. This is because the client cache grows up in case of update. The problem identified in t1 does no longer exist in t2.

TABLE 5 Traversal t3a, t3b, t3c

t3a	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	39,8	47,3	63,6	38,7	44,9
S9 cold	61,3	91,1	92,8	68,3	61,7
M3	832,7	1329,6	805,9	513,8	357,6
M6	1083,9	6467,3	1389,7	1040,0	1064,3

t3b	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	87,4	87,7	122,4	64,4	109
S9 cold	100,8	118,7	140,5	81,0	144

t3c	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	219,0	188,9	284,8	135,6	321
S9 cold	352,6	227,1	318,3	164,6	313

Traversals t3 are the same as t2 except update is done on an indexed attribute. This tests index updates. We can see some weakness of YOODA's index.

TABLE 6 Traversal t4

	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	23,7	12,6	17,0	33,3	10,7
S3 hot	4,7	2,5	1,7	2,4	1,18
S9 cold	23,9	14,4	21,1	48,7	16,7
S9 hot	4,8	2,4	1,7	2,4	1,17
M3	107,0	67,5	91,5	198,4	60,7
M6	107,2	69,2	87,0	258,1	65,2

Traversal t4 is the same as t1 but instead of visiting graph, it visits the document of each composite part. Good performance can be explained by transparent management of long objects in YOODA. They are virtually loaded in a contiguous space. Moreover, the clustering of the document improves performance.

TABLE 7 Traversal t8

	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	1,3	1,4	8,2	1,7	0,3
S3 hot	1,1	0,05	0,5	0,06	0,026
S9 cold	1,4	1,4	8,0	1,8	0,28
S9 hot	1,0	0,05	0,5	0,06	0,026
M3	12,3	5,2	11,4	8,8	3,3
M6	12,2	5,5	11,5	7,9	3,29

t8 traversal scans all the manual associated to each module. It tests how the system can handle long object. Performance is due to the possibility of using standard string functions on persistent long objects without cost overhead.

TABLE 8 Parcours t9

	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	0,2	1,3	8,3	1,2	0,03
S3 hot	0,002	0,002	0,02	0,01	0,001
S9 cold	0,2	1,2	8,0	1,1	0,03
S9 hot	0,002	0,002	0,02	0,009	0,001
M3	0,2	4,9	11,1	1,1	0,88
M6	0,2	4,8	11,0	1,1	0,92

Traversal t9 compares the first and last character of the manual associated with a module. Long object paging enables good performances in YOODA. Indeed, only the first page of a long object is loaded at the first access. Then, only accessed pages are loaded into the client cache.

5.4 Queries

TABLE 9 Query q1

	Exodus	Ontos	Objectivity	Objectstore	Yooda
S3 cold	0,6	2,3	8,4	5,2	0,8
S3 hot	0,007	0,006	0,05	0,01	0,0033
S9 cold	0,7	2,5	8,4	6,0	1,12
S9 hot	0,008	0,005	0,05	0,02	0,0034
M3	0,8	6,6	9,9	6,7	2,83
M6	0,8	9,7	9,6	6,6	2,16

The query q1 simply tests the performance of indexes on a simple access. It looks up ten random atomic parts by identifier. YOODA appears to have the most efficient index. But performance can be explained by the fact that YOODA queries are handcoded.

TABLE 10 Query q2

	Exodus	**Ontos**	**Objectivity**	**Objectstore**	**Yooda**
S3 cold	2,0	4,8	10,8	11,0	2,3
S3 hot	0,008	0,01	0,06	0,04	0,002
S9 cold	5,7	5,0	10,7	14,3	2,57
S9 hot	0,02	0,01	0,06	0,06	0,002
M3	18,0	34,8	33	52,1	19,2
M6	19,1	39,5	37,1	60,5	18,3

Query q2 uses index to obtain objects in a range of keys. YOODA seems to be efficient. This is due to the implementation of index in YOODA. It uses ordered list through key values. A range of keys is obtained by a simple iteration through two positions.

TABLE 11 Query q7

	Exodus	**Ontos**	**Objectivity**	**Objectstore**	**Yooda**
S3 cold	3,2	4,6	17,4	16,7	7,2
S3 hot	0,7	0,1	1,0	0,43	0,07
S9 cold	10,3	6,6	24,5	28,6	11,3
S9 hot	2,2	0,16	1,9	0,4	0,08
M3	31,3	40	100,3	81,3	59,3
M6	31,8	52,6	136,3	90,4	116,4

Query q7 iterate throug every atomic parts. It tests performance of iteration through collections.

TABLE 12 Query q8

	Exodus	**Ontos**	**Objectivity**	**Objectstore**	**Yooda**
S3 cold	8,7	9,4	28,7	21,6	9,49
S3 hot	3,6	2,1	11,2	4,6	0,35
S9 cold	23,4	11,1	35,9	32,4	11,6
S9 hot	11,3	2,1	12,1	4,6	0,36
M3	63,8	87,1	200,6	148,4	68,4
M6	64,7	101,9	227,3	164,4	117,3

Query q8 is a simple join. It finds all pairs of documents and atomic parts where the document id matches the atomic id.

6 Conclusion and Future Work

We have presented here an architecture of an OODBMS based on a model for distribution: the Workspace Model. This work is currently ongoing at APIC systems within the framework of a GIS product. The building blocks of this architecture is the YOODA process. It is a process which contains all the functionalities to handle and share persistent objects.YOODA provides a nested transactions scheme to control distribution. Shared memory is offered as a natural functionality of an OODBMS. CORBA communication is not strongly merged into the model. It is currently in use inside the code of YOODA and available as an independant service for developpers.

YOODA is currently implemented with few simplifications. Only server processes can run activities. Subtransactions are not correctly handled. Every other functionalities of YOODA are available in the current implementation. There is a complete environment which allows industrial use of YOODA. The OO7 benchmark was a good test to validate its usability and its performance.

YOODA futur works are numerous. First, the model must be enhanced. The description of the YOODA process enables many extensions of the Workspace Model. To use these extensions in a consistent way, we must propose evolution of the model. Second, the workspace server must be extended to be able to keep context between two sessions. For the moment, a workspace is not really persistent. Locks and local copies of objects are loosed after disconnection. Object migration is not handle in YOODA. In a distribution context, this functionality must be provided. Lastly, some extensions are desirable to make YOODA a high-level database. Query language or Schema evolution are such extensions.

References

[BAT88] D.Batory and al, “GENESIS : An Extensible Database Management System”, 1988, IEEE Trans on Software Engineering Nov 1988

[CAR90] Carey and All, “The EXODUS Extensible DBMS Project: An Overview”, S.Zdonik, D.Maier (eds): "Readings in Object-Oriented Database Systems" Morgan Kaufmann Publishers 1990 89-102

[CAR93] M.Carey and D.Dewitt and J.Naughton, “The OO7 Benchmark”, 1993, Sigmod Record Proceedings Washington

[CAR94] Carey M.J., DeWitt DJ., Franklin M.J., “Shoring up Persistent Applications”, SIGMOD 94, May 1994.

[COR91] Digital Equipment Corp., Hewlet-Packard Co., HyperDesk Corp., NCR Corp., Object Design Inc., SunSoft Inc., “The Common Object Request Broker: Architecture and Specification”, OMG Document Number 91.12.1, Revision 1.1, Draft 10 December 1991

[DAS89] P. Dasgupta, R. LeBlanc, W.Appelbe, “The Clouds Operating System”, IEEE, 1988

[DEW90] D.J.DeWitt and D.Maier and P.Futtersack and F.Velez, “A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems”, VLDB 1990

[DON94] Didier Donsez, Philippe Homond, Pascal Faudemay, “WEA, A Distributed Object Manager based on a Workspace Hierarchy”, Proc of IFIP Conf. on Applications in Parallel and Distributed Computing, Caracas, Venezuela, Avril 1994, pp247-256.

[KOS86] S.Koshafian and G.Copeland, “Object Identity”, OOPSLA 1986

[LIN87] B.Lindsay and J.McPherson and H.Pirahesh, ”A Data Management Extension Architecture”, 1987, ACM SIGMOD 1987

[LYN86] Lynch N., Merrit M., “Introduction to the Theory of Nested Transactions”, Intl. Conf. on Database Theory, Rome, Italy, September 1986.

[MOS90] J. Moss, “ Working with Persistent Objects: To swizzle or Not to Swizzle”, 1990, Technical Report 90-38 Unversity of Massachusetts t Amherst

[ONT90] Ontos Inc., “ONTOS 1.5 Programmer’s Guide”, 1990, Technical Document

[OBJ92] Object Design Inc., “Objectstore Technical Overview 2.0”, 1992, Technical Document

[RIC90] J.Richardson and M.Carey, “Persistence in the E language: Issue and Implementation”, 1989, Software Practice and Experience Vol 19 Dec 1989

[SHA89] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, C. Valot, “SOS: An Object Oriented Operating System-assessments and perspectives”, Computing Systems, 2(4):287-338, DEcember 1989

[SHE90] E.Shekita and M.Zwilling, “Cricket: A Mapped, Persistent Object Store”, 1990, Proc of the 4th Intl Workshop on Persistent Object Systems 1990

[SCH91] Frank Schmuck, Jim Wyllie, “Experience with Transactions in QuickSilver”, ACM 0-89791-447-3/91/0009/0239, 1991

[SPE89] Spector A, Eppinger J., Daniels D., Draves R., Bloch J., Duchamp D., Pausch R., Thompson D., “High Performance Distributed Transaction Processing in a General Purpose Computing Environment”, In proceedings of the 2nd international WorkShop on High Performance Transaction Systems, Berlin 1989, Springer Verlag, Lecture Notes in Computer Science, Vol 359, pp220-242

[STO90] M.Stonebraker and L.Rowe and M.Hirohama, “The Implementation of POSTGRES”, 1990, IEEE Transactions on Knowledge and Data Engineering Vol 2 No 1 March 1990

[WHI92] S.White and D.DeWitt, “ A Performance Study of Alternative Object Faulting and Pointer Swizzling Stategies”, 1992, VLDB 1992