EOS, an environnement for object-based systems

Olivier Gruber, Laurent Amsaleg, Laurent Daynes, Patrick Valduriez

▶ **To cite this version:**

Olivier Gruber, Laurent Amsaleg, Laurent Daynes, Patrick Valduriez. EOS, an environnement for object-based systems. [Research Report] RR-1499, INRIA. 1991. ⟨inria-00075063⟩

HAL Id: inria-00075063

https://inria.hal.science/inria-00075063v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

EOS, an environnement for object-based systems

Olivier Gruber, Laurent Amsaleg, Laurent Daynes, Patrick Valduriez

To cite this version:

Olivier Gruber, Laurent Amsaleg, Laurent Daynes, Patrick Valduriez. EOS, an environnement for object-based systems. [Research Report] RR-1499, INRIA. 1991. inria-00075063

HAL Id: inria-00075063

https://inria.hal.science/inria-00075063v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France

Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1499

Programme 1
Architectures parallèles, Bases de données, Réseaux et Systèmes distribués

EOS, AN ENVIRONMENT FOR OBJECT-BASED SYSTEMS

Olivier GRUBER
Laurent AMSALEG
Laurent DAYNÈS
Patrick VALDURIEZ

Septembre 1991

Eos, an Environment for Object-based Systems

Olivier Gruber, Laurent Amsaleg, Laurent Daynès, Patrick Valduriez

INRIA, Rocquencourt, BP.105
78153 Le Chesnay, France
E-mail: gruber@nuri.inria.fr

Abstract

Complex data-intensive application domains require a programming environment that should be both more productive and more efficient than the traditional approaches. This translates into three major requirements: efficient support for complex, persistent objects and collections; distribution transparency with control over parallelism; and transaction management. In this paper[1], we present the design of Eos[2] which aims at reducing the chronic mismatch between PL and DBS and avoiding the traditional redundancy between OS and DBS services. The main contributions of Eos are support for: uniform, distributed object management; safe object sharing with distributed, low-overhead garbage collection; high-level mechanisms for dynamic object grouping and computation placement. Eos is currently being implemented at INRIA on a network of Sun3/60 running Mach 3.0

Mots Clés : systèmes d'exploitations, base de données, langage persistant, mémoire virtuelle distribuée, ramasse miettes distribué.

Eos, un Environment pour la Conception de Systèmes Objets

Les applications manipulant de gros volumes de données fortement structurées nécessitent un environnement de programmation plus efficace qui augmente la productivité des programmeurs. Un tel environnement doit répondre aux besoins suivants : un support efficace de graphes complexes et de collections d'objets persistants; la transparence à la distribution sans perte du contrôle sur le parallélisme; et la gestion de transactions. Ce papier présente la conception du système Eos qui vise à réduire le schisme traditionnel entre les langages de programmation et les systèmes de bases de données, ainsi que la redondance de fonctionnalités entre ces derniers et les systèmes d'exploitation. Les contributions primordiales d'Eos sont une gestion d'objets distribuée et uniforme, la sûreté du partage d'objets, un ramasse-miettes peu coûteux, et un mécanisme de haut niveau pour le placement dynamique tant des objets que des traitements. Eos est actuellement en cours de réalisation à l'INRIA sur un réseau de Sun3/60 tournant Mach 3.0.

---

[1] Published in the Hawaii International Conference on System Sciences, Architectural and Operating System Support for Persistent Object Systems Minitrack, 1992.

[2] This work is partially funded by the ESPRIT Project EP2025 EDS.

Eos, an Environment for Object-based Systems

Olivier Gruber, Laurent Amsaleg, Laurent Daynès, Patrick Valduriez

INRIA, Rocquencourt
78153 Le Chesnay, France
E-mail: gruber@nuri.inria.fr

Abstract

Complex data-intensive application domains require a programming environment that should be both more productive and more efficient than the traditional approaches. This translates into three major requirements: efficient support for complex, persistent objects and collections; distribution transparency with control over parallelism; and transaction management. In this paper[1], we present the design of Eos[2] which aims at reducing the chronic mismatch between PL and DBS and avoiding the traditional redundancy between OS and DBS services. The main contributions of Eos are support for: uniform, distributed object management; safe object sharing with distributed, low-overhead garbage collection; high-level mechanisms for dynamic object grouping and computation placement. Eos is currently being implemented at INRIA on a network of Sun3/60 running Mach 3.0

1 Introduction

Complex data-intensive application domains, such as engineering applications or expert database systems, require a productive and efficient programming environment. There are four major requirements of this environment.

1. It should offer an efficient shared memory abstraction over the distributed architectures that for cost and reliability reasons are becoming the dominant computer-based work paradigm.
2. It should offer tools for code reliability and reusability.
3. It should embed an easy-to-use persistence mechanism that supports the transaction abstraction without explicit file system or database calls.
4. It should allow programs to share data easily and efficiently.

Such goals are not new and have been investigated independently by programming language (PL), operating system (OS) and database (DB) researchers. Because of the different perception of the application requirements, each of these communities has only partially addressed these requirements.

- Persistent programming languages [2] and object-oriented database systems (OODBS) [40] address large-scale persistence and code reusability.
- Operating System (OS) projects have focussed on supporting objects distributed in a network of computers, e.g., Clouds [12], Sos [33], and Eden [21]. OS are primarily concerned with the management of large-grain objects (processes, virtual address space, etc.) with few interactions with other objects.

In this paper, we describe an Environment for building Object-based Systems (Eos) which attempts to fulfill the aforementioned requirements efficiently through an integration of PL, DB, and OS techniques. Eos is a persistent, concurrent, object-oriented language over a system-wide, shared, unique object

[1] Published in the Hawaii International Conference on System Sciences, Architectural and Operating System Support for Persistent Object Systems Minitrack, 1992.

[2] This work is partially funded by the ESPRIT Project EP2025 EDS.

space. The language of Eos (Leos) fully abstracts object storage and management issues to the programmer. To achieve performance, high-level and user-friendly mechanism tools permits object and computation placement in order to control object access locality and load balancing. These tools are intended for expert human administrator in a short to medium term, and to be integrated, in a long term, within automated, self-correcting schemes.

The Eos design consists of four major facilities.

1. A distributed single-level store [11, 17] offers efficient support of persistent distributed data and scales up nicely.

   A single-level store also supports fast object addressing schemes and avoids side-effect copies as well as format translations between disk and in-memory formats [11]. A distributed single-level store provides location transparency and supports unconstraint composition of objects.

2. Atomicity and concurrency control at the object granule level allow multiple users to share the object space.

3. An automated garbage collection process permit the unconstrained composition of objects within a shared space.

4. A safe programming language (e.g. well-tamed pointers) provides safe access to the shared object space. Safe means that the object space will remain "consistent" *i.e.* it cannot be corrupted by misbehaving programs.

We have learned the following lessons from our experiences with Eos.

- Integrating DB, OS, and PL functionalities is promising in terms of usability.

- The safe language approach, by offering the ability to manage locks directly in the virtual space, avoids much client/lock-server communication which is costly in terms of messages and context switches.

- Merging the memory coherence and object locking protocols reduces latency and avoids the classical ping pong effect of distributed virtual memory.

- Our distributed garbage collector, taken from previous work by the first author [34], incurs little communication overhead. We extended it to achieve efficient support for dynamic object grouping, by maintaining object grouping even in the presence of topological evolution of the object graph. This grouping mechanism adds small overhead beyond the intrisinc one of the garbage collector.

The paper assumes the reader is familiar with the basic concepts of distributed, object-based programming systems [10], distributed virtual memory [29], and distributed database systems [30]. It is organized as follows. Section 2 gives an overview of Eos's requirements. Section 3 presents the object management in Eos. Section 4 concludes and lists a number of open issues.

2 Eos Overview

In the rest of this section, we make precise Eos requirements in terms of object, distribution and parallelism support.

2.1 Eos Environment

The Eos system attempts to provide an environment for programming over distributed architectures. The environment should be adapted for developing parallelized programs sharing large amount of data. Eos takes two steps towards further integration of three different research fields, namely persistent and distributed programming languages, distributed operating systems, and database systems.

First, Eos merges persistent languages and database systems. The persistency in languages gives a natural, efficient, and safe way to access and manipulate persistent data (as in PS-Algol). Persistent programming languages are a ten-year-old and mature technology and suggest the challenge of redesigning computer architecture to fully exploit the *potential speedup in terms of usability and productivity* without sacrifying performance. This challenge has been sketched out in [3]. Their proposed architecture, adapted in Eos, is depicted in Figure 1 where the Eos layer is typed in bold font. Up to now, Eos has focused on software issues leaving aside hardware enhancements like those achieved in [32, 39].

Applications will be built using a logical persistent languages (LPL). These languages will protect the programmer from physical and storage aspects of the system. They will typically be very high level strongly typed languages, such as those being developed (Napier, Galileo, Machiavelli) and will include appropriate support for bulk and index types. The

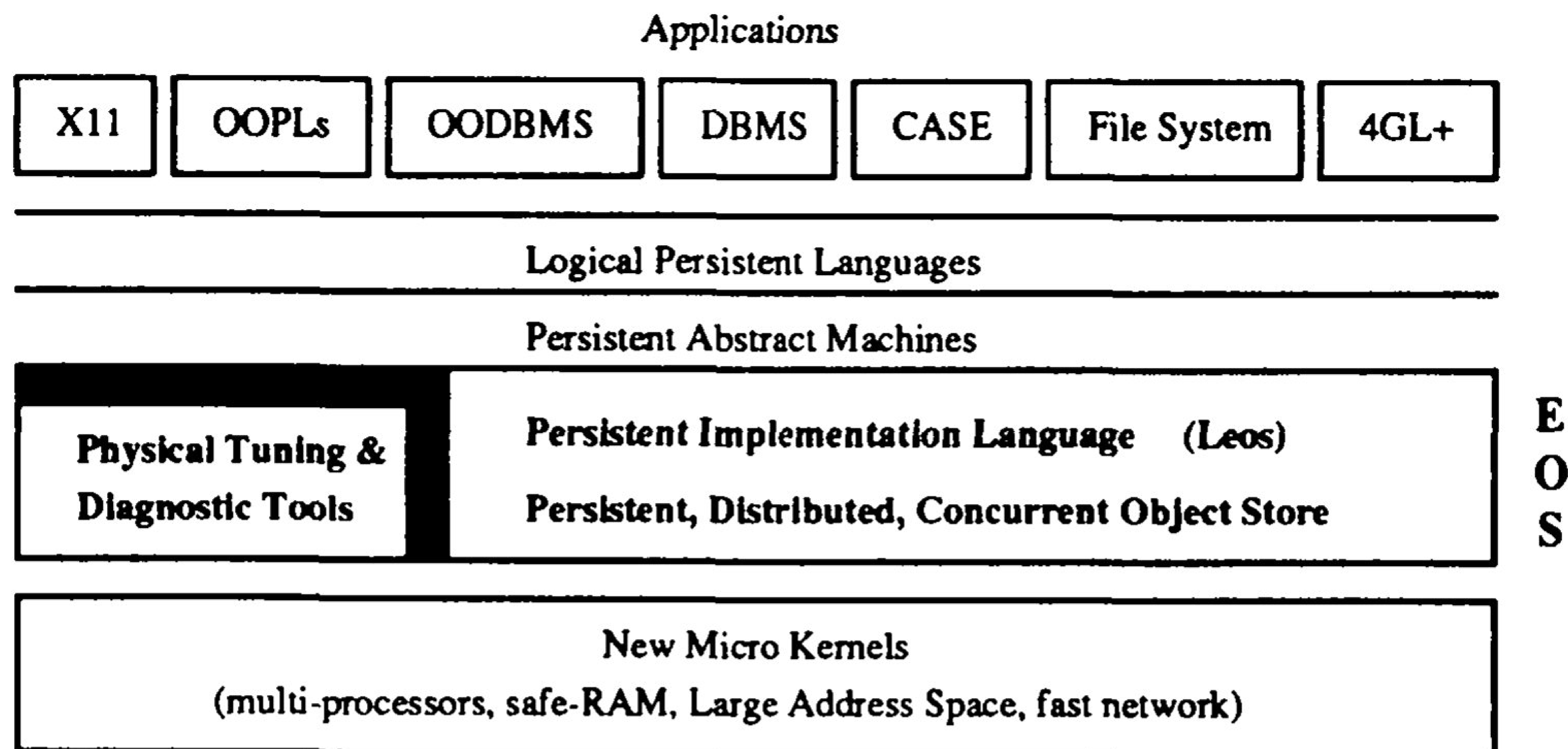

Figure 1: Persistent Computer Architecture

persistent abstract machines (PAM), one per logical persistent language, provide run-time support for the LPL. The PAMs take care of provision of storage of objects of any size and provision of their type system, their protection and concurrency regimes.

Finally, all PAMs are build using the canonical Eos language (Leos). By canonical language we mean the lowest target language in which every programs are ultimately translated within the environment. In other words, Leos is for Eos what is C for Unix.

Leos is a persistent and generic object-oriented language which supports concurrent and parallel programming. By parallel programming we mean multithreading of a program in order to improve performance while concurrent programming means multiprogrammation over a shared object space. We now detail briefly the main issues related to the object management.

Uniform Persistence. Persistence is the ability of objects to outlive their creator scope. Absolute transparency is necessary to benefit fully of the advantages of persistence. Hence, persistence should be orthogonal to type, instanciation, and should be propagated automatically to sub-objects of persistent objects as defined in [1]. A naming service, called the *catalog* is provided and constitutes the persistency root. Objects directly reachable from the catalog are called *named objects* and are persistent. Objects reachable transitively from the named objects are persistent.

Genericity for user-defined collection support. Leos should be able to support any kind of data structure. This suggest a form of genericity as in Eiffel [27]. Genericity enables to write generic, user-defined data structures such as lists, hash tables or Btrees without requiring untyped pointers and casts as in C. Therefore, genericity with the needed run-time checks enables the language to stay safe, for e.g. to deal with the contravariance problem of object-oriented type system.

However, type-safety is not sufficient, *safe object sharing* is also necessary. *Safe object sharing* means that the system guarantees the absence of dangling pointers. A dangling pointer contains an incorrect address, for instance, of an object which no longer exists. Safe object sharing is achieved by forbidding casts, undistinguished unions, pointer arithmetic and explicit freeing of objects. Therefore, Eos should provide an automated garbage collection. As cycles in the object graph arise in pratice and might dangerously waste ressources in a persistent environment, the collector has to be of the marking family.

Parallel programming. Leos offers explicit rather than automatic parallelization. However, automatic parallelization and optimizations of end-user (human) programs is possible at the levels above Leos, in the PAMs or LPLs. Methods in Leos can be invoked asynchronously or synchronously. A synchronous method invocation forces the invoker to suspend until the method returns. An asynchronous method call lets the invoker run in parallel with the invokee. Standard synchronization and communication tools such as barriers or rendez-vous and ports, are supplied. By default, a synchronous paradigm is adopted.

Concurrent Programming. Database systems pro-

vide global schema management and concurrency control in order to manipulate a shared space of objects. Much work is still to be achieved with respect to schema management, but, concurrency control by contrast is a mature technology. Each Eos program is a transaction with nested sub-transactions. Consequently, programs are atomic, insulated, and resilient. Serializability is enforced by a *lock-safe* two-phase locking while atomicity and resiliency are enforced by logs. *Lock-safety* ensures that objects are always locked, and further locked in the correct mode before they are used. Lock-safety is ensured by the compiler which takes care of the two-phase locking and committing of atomic methods by inserting lock requests and commit calls. Nested transactions (as in Argus [26]) ensure site independancy with respect to failures when programs are distributed.

However, serializability is often too strong and needs to be relaxed in order to provide a higher degree of concurrency. We name such relaxed concurrency controls *semantic concurrency control* since they rely on higher-level semantic. On the one hand, highly shared data structures such as collection indices have long been identified as bottelnecks when managed with strict 2PL and should be managed with specific concurrency control methods [6, 22]. Recall that collections are user-defined in Leos through the genericity of the language in order to cope with specific needs. Therefore, such specific algorithms should be expressable in Leos. On the other hand, parallel computations requires non serializable executions on structures supporting the parallelism. For instance, parallel computations needing to exchange informations using a shared buffer do not respect a serializability criterion.

The support of semantic concurrency control has two implications : the granule of locking should be the object and the system should offer a specific unlocking facility. Object locking granule is unavoidable since page locking causes unpredictable concurrent behavior. For example, given a list of four objects A, B, C, D, and two transactions, t1 reading the list starting from A, and t2 modifying the list starting from C. There will be no deadlock if object locks are used. There may be if page locks are used and if A and D are in the same page. The unlock ability rationale is straight forward since, for instance, lock-coupling requires to be able to unlock an object. The unlock call applies to a class instance (a sole object), releases the lock and ensures atomicity as durability.

The next final step Eos takes is to integrate achievements of distributed programming languages [9, 36, 8] and operating systems [28, 33, 31]. Distribution should be abstracted. This is achieved by a shared, distributed object space which hides the distributed nature of the architecture. However we separate such a transparency for the programmer from control of object placement and load balancing for the administrator.

Administration is actually a human task which requires high-level tools and much usability but should be ultimately automated. Human are notably poor at grasping dynamic behaviors of systems [38] and therefore, efficiency of human administration is likely to be limited. Consequently, administrators should be helped in their task by high-level mechanisms [5]. However, a long term challenge of Eos is to provide automated, self-correcting administration. We believe that persistent architectures as ours are promising for this purpose due to their integration of the different layers usually involved in object placement (language, object manager, OS).

3 Eos Object Management

We now focus on the solutions to distributed object management since this remains the most challenging obstacle to Eos's objectives. Eos's distributed object management relies on three known techniques : nested transactions, distributed virtual memory, and garbage collection. However, it is difficult to combine them in order to fulfill our requirements : support for small objects (dozens of bytes), object granule locking, safe object sharing (recal that it requires a mark-based collector to reclaim cycles in the object graph), large-scale persitence, and of course efficiency!

3.1 Single-level Store

Eos adopts a *distributed single-level store* approach and relies on the memory object facility provided by the Mach microkernel. In Mach, a task can associate (i.e. map) a given region of their address space to a memory object using the vm_map kernel call. After doing so, the memory object plays the role of an external pager for this region, i.e. it is called by the kernel whenever a page in the mapped region needs to be read or written to disk. A memory object is a user process and communications with the kernel takes place through a port. Hence, it is possible to tailor the virtual memory management to the user needs. The shared object space is supported by a distributed memory object which is identically mapped in all Eos task. This ensures the system-wide validity of pointers.

The architecture of the distributed object space is depicted in Figure 2. Pages are duplicated on faults and a specific object-based consistency protocol maintains the page coherence (see section 3.3). In this context, one delicate issue is to locate pages to service fault without incurring huge location directory. Several schemes exist, centralized directory, static ownership which associate statically pages to processors, dynamic ownership which allows pages to move around and uses a forwarding mechanism to find the actual page location [24].

We adopted a novel approach which benefits from our administration tools. The space is splitted into ranges and each node may require ownership for a given range. A *node* is a uni- or multi-processors sharing a local physical memory and some disks. Range ownership is static, so the owner of a page does not change overtime. Hence, location directory is limited to range ownership instead of page ones which is far smaller and stable. Range ownership is managed by a centralized service which does not constitute a bottleneck since range ownership evolution should be time-sparse. We do not suffer from the usual load-balancing limitations of static ownership since load balancing is achieved by reclustering objects within pages in accordance with the administrator hints, report to section 3.2.

The object space is also structured to take into account its distributed nature in order to enhance its management, i.e. garbage collection, allocation and compaction. The object space is divided into containers which in turn are subdivided into clusters. A container is a set of clusters contained on a single node which are insulated from the rest of the object space by two indirection tables. In other words, all incoming and outgoing references of a container go through an indirection table. The two tables are called the *In-Reference Table* and the *Out-Reference Table*. A container is local to a node, called its *owner node*, but a node may hold several containers.

The main purpose of containers is to optimize the garbage collection process. Garbage collecting a large-scale persistent and distributed object space raises several problems. The potentially high number of live objects in the persistent object space forbids a global scan of the object space and also precludes the use of copying or compacting collectors. First, a global scan is unpracticable because it would last too long. Furthermore, it would be extremely costly over a distributed space incurring high network traffic and further requires a global termination detection. Such a system-wide synchronization reduces the efficiency of the collector by stretching the process duration. Moreover, it is non-scalable and fragile wrt node crashes. Second, a copying or compacting collector on a large-scale heap incurs an unacceptable byte-copy time, but it gets worse when used on a persistent heap since it yields many disk access to save the new heap state.

Containers enable incremental, node-local garbage collection. Periodically, containers are garbage collected independently. Container-local marking is conservative considering the In-Reference Table as the root. Remark that the collection process is local to a node since containers are themselves local to a node. Local collections of containers clean the Out-Reference Tables and thus enables to better estimate the In-Reference Table of pointed-to containers in a transitive way [34, 20]. Moreover, the overall collection process does not entail any kind of system-wide synchrony. Distributed cycles are reclaimed in background as a side effect of the In-Reference Table estimation [16].

Furthermore, Eos garbage collector is non-copying and on-the-fly. This process detects garbage, marks as free the corresponding memory chunks but neither copies nor sweeps the object space. On-the-fly (parallelized) collection [13] nicely fits the requirements of interactive applications. Furthermore, it can be quite efficient on multi-processor workstations like the oncoming Luna or Snake ones.

A non-copying collector gives rise to the question of how objects are allocated and memory reorganized? This is one purpose of *clusters*. A *cluster* is a set of virtual pages, possibly non-contiguous. A cluster belongs to a single node called its *owner* node. A cluster has a root composed of a set of objects which are reachable from outside the cluster, i.e. from other clusters. Objects are allocated within clusters in accordance to the object grouping strategy defined by the administrator, see section 3.2.

The object allocation scheme borrows from [23]. The basic idea is to benefit from the existing slicing of the virtual space into ranges used to reduce the size of page location directory. The concept of ownership is extended to encompass allocation right. A node allocates objects linearly within virtual pages themselves allocated from the ranges it owns. An object (class instance) never spans page boundaries[3]. Therefore, object allocation is fast without any distributed synchronization.

Before we detail the hole coalescing mechanism, we need to say a word about object identity. Object identity is supported through virtual memory addresses

[3]Recall that the PAMs are responsible of the provision of objects of any size

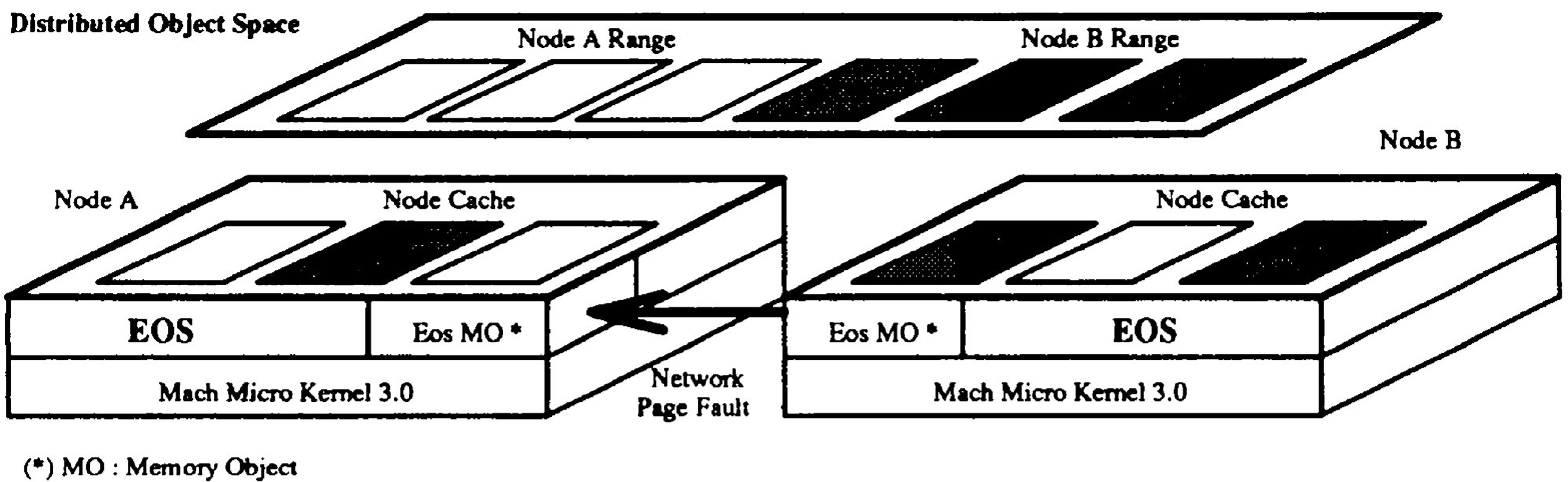

Figure 2: Persistent Computer Architecture

and a forwarding mechanism. In other words, Leos pointers, called *object-oriented pointers* (oop), directly point to objects. When an object is moved in memory, a *forwarder* is left in place of the old object location. A forwarder is a *temporary* system stub which forwards the dereferencing of an oop towards the new location of the object. Forwarders are managed using a jump strategy [14] and thereby can be reclaimed asynchronously by the garbage collector.

Clusters provide a satisfactory answer to hole coalescing problem within a large-scale persistent heap. To cope with the nature of the heap, compaction is achieved on a per cluster basis. So, the byte-copy time and the number of disk accesses are acceptable. A scavenging process is used to pack clusters since it copes with our object identity scheme and further group objects within virtual pages. The cluster root is used as the scavenging root. It copies all objects of the cluster (from-space) within a minimum set of new pages (to-space) leaving forwarders in place of the old object locations. Pages of the from-space can be freed when all their forwarders have been reclaimed, so they can be reused to allocate new objects.

Cluster compaction has another positive side-effect on node locality of clusters. Distributed allocation of objects tend to scatter cluster pages on ranges owned by different nodes. This hurts locality of treatments placed at the owner node since the cluster use induces network page faults. Cluster compaction, by doing a complete copy within newly allocated pages which belong to the ranges of the cluster owner node, reforms a truly local cluster.

Clusters also enhance caching performance. On the one hand, the oop scheme using *temporary* forwarders ensures a better average locality during oop dereferencing compared to those providing logical identity based on global object table. On the other hand, a cluster maintains a bitmap holding the color of its objects (dead or alive), so the garbage collector marking phase does not dirty all pages of clusters but only their system structures. This greatly reduces the amount of dirty pages and consequently paging activity.

A last point to discuss about single-level store is how persistency is supported? The most efficient scheme to support uniform persistence at the language level is undoubtly a persistent virtual memory [37]. However, we adopted the durability concept of transactions rather than simple memory resiliency. Resilient virtual memory is not new [37, 25] and relies on checkpointing techniques. Checkpointing approaches make the object space progressing from one stable state to another. However, this does not support short transaction durability. Instead of supporting it through some special mechanism, we believe that better performance can be achieved using a single mean. We are currently studying the positive effects of safe RAM which would permit to consider fully asynchronous checkpoints as the aforementioned ones.

3.2 Object Space Administration

Object space administration concerns the placement of objects and computations in order to achieve sufficient locality for the distributed single-level store to work. This section presents our mechanisms. We will first present our mechanism to group object within clusters, then show that it also applies to node placement of objects around a network, and finally introduce our scheme for load balancing.

3.2.1 Object Grouping

Object grouping is a twofold issue, on the one hand the administrator should be provided with a mean to express his or her knowledge about the correct grouping, and on the other hand, the system has to implement the grouping with small overhead.

The administrator expresses the relevance of links between objects in the schema, i.e. oop members of every class are labelled with a relevance. By default, loose relevance is used. Hence the grouping behavior is encapsulated within the class. However, relevance setting on a per instance basis is also supported. The link relevance has three levels, tight, loose, and free.

- Tight links represent primary access paths to an object, i.e. highly-used references.
- Loose links are secondary access paths, i.e. less frequently-used references.
- Free links represent rarely-used access paths.

Tight and loose links correspond to the usual concepts of primary and secondary access paths, i.e. data are clustered in accordance with their primary access path. But the concept of free link is new. Tight and free links are converse; a tight link suggests to group the two linked objects while a free link suggest their separation to the benefit of other loose or tight links. Without the free relevance, such a reclustering would have required from the user to locate another parent and to set its link to a tight relevance.

Given the indication of the relevance of links between objects provided by the administrator, Eos groups objects accordingly within clusters. First of all, objects are created near their first parent, i.e. in the same cluster[4]. Once shared, an object is grouped with its parent which has the most important relevance. If it is not unique, then the child is non-deterministically grouped with one of these parents. This is achieved asynchronously as a side-effect of the marking phase of the garbage collector. Indeed, it walks through the entire object graph and therefore is aware of all parents and their respective relevance. It is important to point out that this approach allows Eos to keep object grouping accurate in presence of topological evolution of the object graph with very small overhead beyond that required by garbage collection itself.

The next point to consider is how clusters accomodate this object grouping. Clusters are variable-size recipients (set of virtual pages) in which objects can be added or retrieved. Though, cluster size should be kept small since their main rationale is to permit a localized hole coalescing. So, clusters split when a splitting threshold is reached and are freed when they get empty. How small should be that threshold is a tuning value that requires more experiments to be known.

Cluster splitting is a graph partitioning problem since the goal is to minimize the cutset. Indeed, the lower the cutset, the better will be the locality within clusters. However, existing solutions [19, 18] are too slow. Our approach is to translate the problem into a tree splitting problem which is simpler and faster (with linear complexity). The basic idea is to use link relevance to map the general graph into a tree by only keeping the most relevant link to a shared object.

The rationale behind the free-at-empty behavior of clusters is that finding another cluster to merge with is delicate in regards of the cutset, and that likely, the two behaviors would be equivalent as it has been shown for Btrees in database systems [35].

3.2.2 Node Placement of Objects

The previous placement mechanism extends nicely to node placement of objects. Indeed, objects can be also moved between containers according to relevances of links. So, this object migration between containers enables the administrator to control container connectivity. This is good for at least two reasons. First, a strong connectivity induces some spatial and time overhead due to the indirection tables. Second, inter-container links may go accross the network when the containers rely on different nodes, so a strong connectivity may increase network traffic.

As consequence of object migration between containers, they shrink or grow as their clusters do, and so they may split. Container size should be maintained small enough so garbage collection stays efficient. Containers can be split using traditional graph partitionning techniques [19, 18] since their cardinality (number of clusters) and splitting frequency are much smaller than the cluster ones.

3.2.3 Load Balancing

Load balancing is important to reduce network traffic and fully exploit the parallelism possibilities of the architecture. Load balancing is twofold. One fold concerns data and computation placement in order to increase locality of object reference and the other to equilibrate the container size managed by the different nodes to keep garbage collection feasible and to limit

[4] In Leos, object creation is achieved by sending the create() method to the oop within the first father, therefore it is known at creation time.

the network page fault servicing. Eos provides two mechanisms for this purpose. The first trivial one is the ability to move containers around the network to match inter-container access patterns, reducing network traffic.

The second is the ability to play with both function and data shipping paradigms. Function and data shipping paradigms are adapted to disjoint access patterns [15] and therefore should both be supported. The default paradigm is data shipping since this corresponds to the distributed single-level store philosophy. It is efficient when locality is high and clustering adequate.

Function shipping paradigm can be enforced by the administrator which expresses in the schema that a method has to be shipped towards its receiver. Shipping methods yields a server-like behavior. This is efficient for treatments with high selectivity, i.e. using small percentage of clusters, or high update or creation rates. Hence, the class encapsulates the behavior of object-oriented message passing wrt distribution. Provided with the two paradigms, the administrator is free to equilibrate its systems between fully distributed single-level store or full client/server behavior, thereby, controlling load balancing with a sufficient precision.

3.3 Coherency Protocol

Distributed virtual memory and related coherency protocols have been the subject of much research [29]. To increase parallelism, virtually all systems replicate data. Two types of protocols handle data replication : write-update and write-invalidate. In a *write-invalidate* protocol, there can be many copies of a read-only object, but only one copy of a writable object. The protocol invalidates all copies of an object except one before a write can proceed. In a write-update scheme, however, a write updates all copies of an object.

Most distributed systems have write-invalidate coherency protocols. Li and Hudak [24] show that the write-invalidate protocol performs well for a variety of applications. Subsequent research indicates that appropriate hardware can support write-update efficiently [7]. The main advantage of write-update protocol is to avoid the ping pong effect of write-invalidate ones. A ping pong effect arise when two nodes compete for a page and invalidate each other copy repeatedly.

However, memory coherency protocols do not support concurrency control, but only ensure page consistency. This two-layer architecture induces hurting overhead when the locking granule is the object. Coherency protocols are unable to manage consistency at the object granule since objects are higher-level entities unknown to the virtual memory layer. Such protocols usually ensures page consistency and sometimes minipage consistency using specialized hardware. So, double work occurs since an exclusive lock on an object has to be acquired before a writer can proceed, and later, the first update of the object will cause the virtual memory layer to ensure page consistency. This incurs unnecessary messages and latency.

Moreover, this approach does not benefit at all of lock semantic to enhance performance. While a write lock is acquired for the duration of an entire transaction, often encompassing several updates on a single object, memory coherency protocols still ensure consistency on a per update basis. For instance, the write-invalidate protocol may loss the page writability due to updates made to other objects within the same page (ping pong effect); the write-update protocol propagates unnecessarily each updates while it is needed at transaction-commit time only.

Eos proposes a new tack which merges the philosophy of the write-update protocol and lock semantic at the object granule. We first introduce the architecture, and then present our protocol.

3.3.1 Architecture

Recall that each page has an owner node and this association is static. On the occurence of a page fault at some other node than the owner one, a copy of the page is requested to the owner node.

Locks upon objects of a given page are granted by the *manager* node of the page. The association between a page and its manager node is dynamic, so, each page has a static owner node and a dynamic manager node, which may or may not be identical. The manager of page P is the node which first request the page P to the owner node of P. The advantage of this dynamicity of the manager is to ensure local management of locks whenever there is a single copy of a page, whichever is the node using it. The manager keeps track of the different copies of the page around the network. Nodes having a copy of a page are called *copy-holder* nodes.

3.3.2 Eos Protocol

The basic idea of Eos protocol is to be *slightly* optmistic for locking, i.e. to grant locks on locally available information. Local information are updated by gossip messages between copy-holder nodes. A *gossip* message is an asynchronous background message.

Here follows the description of our protocol. Let start with a single copy of a page P, and some transactions running on the manager node of P. The locks are granted and managed locally directly in virtual memory. For each lock, the manager maintain two information. One, the current *lock state* :

- free, grantable to anybody.
- read, grantable to readers only.
- exclusive, grantable to nobody.

Two, the list of pending transactions. This list is composed of *waiting groups*. A waiting group is one or more transactions which await for a given state of the lock and can be scheduled altogether. A waiting group can be composed of one or several readers (read group, noted RG), or one writer (exclusive group, noted XG). An example is depicted in Figure 3.

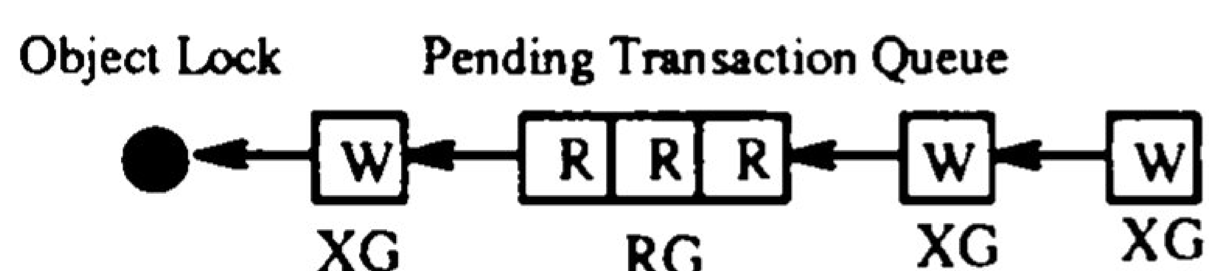

Figure 3: Pending Waiting Group Illustration.

When pages get duplicated, each copy-holder node locally maintains the lock states of objects which rely within pages currently in its cache, and the corresponding pending queue of local transactions. First of all, a copy-holder node gets along with a page copy the object lock states, known at the page manager node when it serviced the fault. Then, the copy-holder manages the locks as shown in Figure 4. One can point out that the concept of waiting groups reduce the amount of gossiping. Gossiping is further reduced because copy-holders do not gossip the granting of a lock in read state to a new reader. Both otpimizations are especially interesting if readers dominate and parallelism is high.

The local state of the locks and enqueuing policy are subject to external control from the manager. First, the lock state can be changed, i.e. the manager indicates that the lock state has changed due to other copy-holder activity. For instance, a reader got a lock at some other node that was believed free locally. Second, the manager can force the creation of a new waiting group for next requesters of a given lock, even if they are compatible with the last group of the queue. This is achieved by a *force_wait* gossip and is motivated later.

```
if ( queue is empty ) {
  switch ( lock_state ) {
  free :
    grant the lock & gossip to page manager.
  default :
    if ( compatible(requested_mode, lock_state) )
      grant the lock.
    else {
      create a new waiting group,
      and gossips that to the page manager.}
  }
}
else {
  if ( compatible(requested_mode, tail_group) )
    add the requester to the last waiting group.
  else
    create a new waiting group, and
    gossip that to the page manager.
}
```

Figure 4: Copy-holder Lock Management.

From the copy-holder gossips, the manager builds a global knowledge of the state of locks and the pending waiting groups. Recall that each copy-holder informs it of the locks it grants and the waiting groups it enqueues. This global knowledge serves two purposes. First, it allows to detect misbehaviors of copy-holders because of optimism. Second, it enables to control the re-scheduling of waiting transactions.

Misbehaviors are easily detected at the manager. The manager keep the lock state accordingly to the gossiped informations from the copy-holders. Furthermore, it forwards meaningful transitions to all copy-holders. When the manager receives a lock state from a copy-holder which is incompatible with its own lock state, it simply discards the message. Indeed, the misbehavioring copy-holder will later receive the gossip message about the manager state of the lock, detect the conflict, and abort the corresponding transaction. The fact that the manager discards the message ensures that only one node will detect the violation and cause an abort.

Misbehaviors happen because of network latency, so two copy-holders may grant the same lock in incompatible modes. The scheme is the following which is depicted in Figure 5. A page P is duplicated at two copy-holder A and B. Copy-holder A locks an object O in page P in a α mode and gossips the lock transition to the manager. The manager receives the gossip and accepts the transition because it does not conflict with the object lock state, and then, gossips the new state

of the lock to copy-holder B. Meanwhile, B granted a lock on O in a β mode which is incompatible with the α mode and gossiped this fact to the manager. Then, the manager will get a gossip that shows that B got wrong, and simply discards it. Node B later receives the manager gossip about the α state, discovers its misbehavior, and aborts the corresponding transaction.

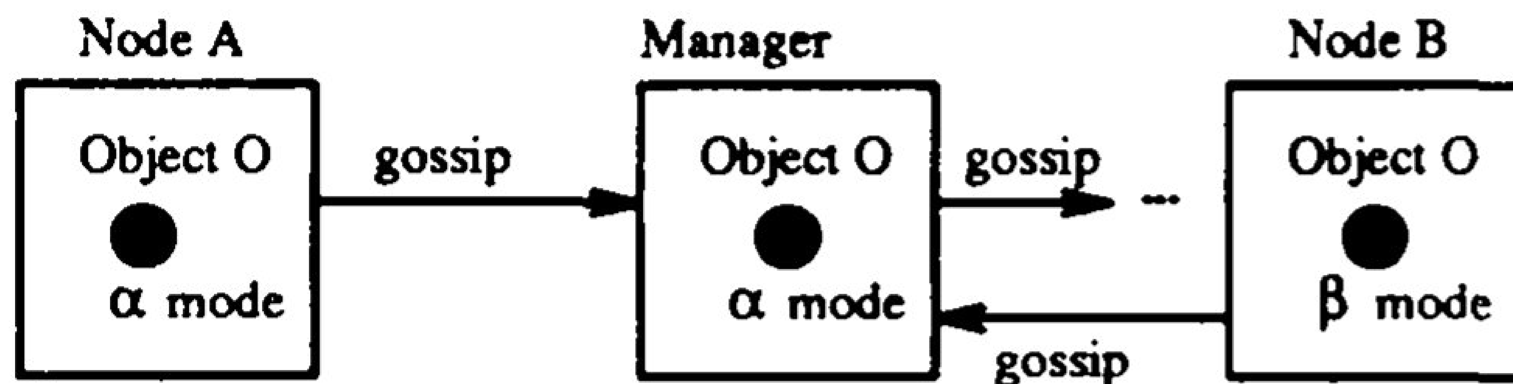

Figure 5: Faulty Optimistic Behavior.

However, our gossip-based protocol is only *slightly* optimistic and should almost induce the same abort rate than the pessimistic RPC-based one. First, a copy-holder does not misbehave systematically on new copy pages since lock states are piggy backed on page transfers. Second, locally information are out-of-date during a small period of time only. This time interval is called a *window*. Finally, the nested nature of our transaction model makes aborts cheaper.

The window is the period of time during which a copy-holder is unaware that another copy-holder changed the lock state of a common object. The window average size is about a few milliseconds on Ethernet since it incurs at most two gossip messages. So, the window size is small compared to transaction duration. Therefore, the usual negative effect of optimism, i.e. increased abort rate, should be almost avoided. This stays true even if the window size is enlarged because of buffering of gossip messages. Buffering reduces network traffic.

The manager role is also to control the transaction scheduling so to avoid starvation and ensure correctness. The above protocol does not avoid writer starvation. Starvation may appear on an object O if over several copy-holder nodes, new readers always get the lock while a writer is waiting. The manager is aware of its presence, but not the other copy-holders. So the manager has to gossip a *force_wait* message in order to suspend incomming readers at all copy-holders. This generalizes to : a manager gossips a *force_wait* message when it is enqueuing a new waiting group.

Transaction scheduling also encompass the awakening when locks are released, this is the commit protocol. We will not consider fault tolerance, albeit its importance, for the sake of simplicity. So, committing a transaction is equivalent to release its locks and ensure memory causality. Before the commit can start, the transaction has to confirm all its locks by requesting an acknowledge from all the managers concerned, i.e. the managers of the pages which hold the objects locked by the transaction. An acknowledge is sufficient because the Mach messages are not lost and point-to-point ordered. Once these acknowledges have been received, either the copy-holder is aware of its misbehavior or all locks are confirmed.

Once locks are acknowledged, the commit can start. The read and write locks should be released and the corresponding state transitions should be gossiped to the manager which will forward them to all other copy-holders. In the case of write locks, causality has to be ensured before they can be released. I.e. the copies of the pages has to be brought up to date with the object after-images. The committing node sends these after images to the manager node and then releases its local write locks, but do not schedule any pending transaction. Remark that if a lock has no pending transactions, it is locally grantable. The manager updates its own copy of the pages with the after images and releases its locks. Then it forwards the after images to all copy-holders and informs them of the new lock states. For objects having no pending transactions, it forwards the free state. For others, it grants the lock to its first waiting group, and forwards the scheduling information to the concerned copy-holders.

We choose to ensure object-based causality versus page-based causality for a number of reasons. First, the ratio between reads and updates is likely to be large. Second, the administrator is supposed to place data and operation accordingly, hence operations performing intensive updates or object allocations should mostly execute at the manager node. Finally, this solution trades-off CPU cost (for extracting objects) for communication cost (only updated objects are sent over the network). This is encouraged by the hardware trends which make processor power increasing far more rapidly than network bandwidths, as the apparition of multi-processor workstations.

4 Conclusion

We presented in this paper the Eos project, actually ongoing at INRIA in the Sabre project. Eos attempts to define, design, and prototype a more productive and efficient programming environment. Eos targets distributed architectures, and data-intensive applications which concurrently access shared data. To fulfill

this goal, Eos integrates existing technics of PL, DB and OS, and proposes a uniform architecture and design. Eos promises much gain in term of performance and usability.

Object management in Eos is fully designed and a first prototype is running. However, since the object management issue is the most challenging obstacle to Eos's objectives, complete measurements are necessary to fully evaluate the resulting system. In peculiar, transaction commit should be evaluated with great care and optimized accordingly. We believe that safe RAM is a very promising hardware support for this purpose. The overhead of garbage collection and its associated grouping mechanism has to be measured in order to decide if non-stop systems can be supported. More generally, Eos ability to scale up should be confirmed, in particular, its ability to support database applications.

Eos futur works are numerous and exciting. First of all, the language has to be precisely defined. The challenge is to integrate the numerous ideas designed to enhance distributed object management within a uniform and usable data model. Issues of the compilation process are already mastered within different existing languages. Along with the language, the administrator interface to tuning and diagnostic tools need to be defined. We envision to reuse a previous work conducted in our project on object-oriented schema management within persistent systems [4]. The advantage of the distributed nature of our architecture for better fault tolerance should be explored. Finally, the automated, self-correcting administration goal requires a fine study and analysis of Eos under various application patterns and load.

Acknowledgements We appreciate the comments from the anonymous referees. We especially thank Dennis Shasha for its tenacity in helping us to clarify this paper. We appreciate the fruitful discussions with M.J. Bellosta-Tourtier and Eric Amiel. We are also grateful to Marc Shapiro for its interest in Eos and for our discussions.

References

[1] M. Atkinson, P. Bailey, K. Chrisholm, K. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), 1983.

[2] M. Atkinson and O. Buneman. Type persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.

[3] M. Atkinson and R. Morrison. Persistent system architectures. In *Persistent Object Systems*, 1989.

[4] M. Bellosta-Tourtier, F. Viallet, and P. Valduriez. Design considerations for OMNIS, an object management interface system. In *Proc. of the TOOLS'91 Int. Conf.*, Paris, March 1991.

[5] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store : Clustering strategies in O2. In *4th Int. Workshop on Persistent Object Systems*, Martha-Vineyard, Mass., September 1990.

[6] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] R. Bisiani and M. Ravishankar. Plus: A distributed shared-memory system. In *Proceeding 17th International Symposium Computer Architecture*, 1990.

[8] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE*, SE-13(1), January 1987.

[9] J. Chase, F. Amador, E. Lazowska, H. Levy, and R. Littlefield. The Amber system : Parallel programming on a network of multiprocessors. *Operating Systems Review*, 23(5), December 1989.

[10] R. Chin and S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.

[11] G. Copeland, M. Franklin, and G. Weikum. Uniform object management. In *Proc. of the Int. Conf. on Extended Database Technology*, Venice, Italy, 1990.

[12] P. Dasgupta, R. LeBlanc, and W. Appelbe. The Clouds operating system. *IEEE*, 1988.

[13] E. Dijkstra, L. Lamport, A. Martin, and C. Scholten. On-the-fly garbage collection : an exercise in cooperation. *Communications of the ACM*, 21(11), November 1978.

[14] R. J. Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proc. of the Fifth SIGACT/SIGOPS Conf. on the Principles of Distributed Computation*. SIGACT/SIGOPS, August 1986.

[15] A. Hastings. Distributed lock management in a transaction processing environment. Technical report, CMU, Pittsburg, May 1989.

[16] J. Hughes. A distributed garbage collection algorithm. *Language and Computer Architecture*, 1985.

[17] B. Kotch and al. Cache coherency and storage management in a persistent object system. In *Implementing Persistent Object Bases*, 1990.

[18] B. Krishnamurthy. An improved min-cut algorithm for partitionning VLSI networks. *IEEE Transactions on Computers*, C-33(5), 1984.

[19] P. V. Laarhoven and E. Aarts. *Simulated annealing : theory and applications*. Mathematics and its applications. Reidel, 1988.

[20] R. Ladin and B. Liskov. Highly-available distributed services and fault-tolerant distributed garbage collection. *In Proc. of the 5th Symp. on the Principles of Distributed Computing*, August 1986.

[21] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The architecture of the Eden system. *In Proc. of the 8th ACM Symp. on Operating Systems Principles*, December 1981.

[22] P. Lehman and S. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4), December 1981.

[23] K. Li. IVY : a shared virtual memory system for parallel computing. *In Proc. of the Int. Conf. on Parallel Processing*, August 1988.

[24] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4), November 1989.

[25] K. Li, J. Naughton, and J. Plank. Real-time, concurrent checkpoint for parallel programs. Technical Report CS-TR-239-89, Department of Computer Science, Princeton University, Princeton, December 1989.

[26] B. Liskov and R. Scheifler. Guardians and Actions : Linguistic support for robust, distributed programs. In *ACM Transactions on Programming Languages and Systems*. ACM, July 1983.

[27] B. Meyer. *Object-Oriented Software Construction*. Computer Science. Prentice Hall, 1988.

[28] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[29] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 1991.

[30] M. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 1991.

[31] D. V. Pitts and P. Dasgupta. Object memory and storage management in the Clouds kernel. In *The 8th International Conference on Distributed Computer Systems*, pages 10–17, San José, CA(USA), June 1988. IEEE.

[32] J. Rosenberg, D. Koch, , and J. Keedy. A capability-based massive memory computer. In *Persistent Object Systems*, 1989.

[33] M. Shapiro, Y. Gourhant, S. Habert, L. Mosseri, M. Ruffin, and C. Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[34] M. Shapiro, O. Gruber, and D. Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Technical Report INRIA-1320, INRIA, Rocquencourt (France), November 1990.

[35] D. Shasha and T. Johnson. A framework for the performance analysis of concurrent B-tree algorithms. *In Proc. of the ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, Nashville*, March 1990.

[36] S. Shrivastava, G. Dixon, and G. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, January 1991.

[37] S. Thatte. Persistent memory for symbolic computers. Technical Report TR-08-85-21, Central Research Laboratories, Texas Instruments, Incorporated, Dallas, July 1985.

[38] G. Weikum, C. Hasse, A. Monkeberg, and P. Zabback. The COMFORT project : A comfortable way to better performance. Technical Report 137, Computer Science Department, ETH Zurich, Switzerland, July 1990.

[39] I. Williams and M. Wolczko. An object-based memory architecture. In *Implementing Persistent Object Bases*, 1990.

[40] S. Zdonik and D. Maier. *Readings in object-oriented database systems*. Morgan Kaufmann series in data management systems, San Mateo, CA, 1990.

ISSN 0249-6399