A Causal Distributed Shared Memory Based on External Pagers

F. Boyer
Unité Mixte Bull-IMAG/Systèmes, 2, avenue de Vignate, 38610 Gières,
France - Internet: guide@imag.fr - Phone: +33 76634848

Abstract:
The objective of this paper is to propose an efficient implementation of a distributed shared memory based on the Mach External Pager facilities. We concentrate on memories where objects are replicated among nodes, which requires a mechanism for managing the consistency of the copies of an object. Because usual consistency protocols based on strict consistency may lead to a high message traffic in the system, we propose to use a released form of consistency which allows reads and writes to be executed in parallel on copies of a given object. Released consistency allows a significant reduction of the number of messages, which leads to increased concurrency and better performance compared to usual strict protocols. However, as we want to make the implementation transparent to the programmer, some dependency management must be added to the released consistency in order to make the read operations return consistent values with respect to causally related operations. A *causal memory* is a memory respecting the causal dependencies between operations. Such a memory is less consistent than a centralized memory. However, it provides enough consistency for distributed applications in which communication between processes is achieved through shared data.This paper proposes a simple and efficient implementation in which both the released consistency protocol and the causal dependencies management are done by the External Pagers. We show that the cooperative architecture of pagers exactly fulfils causal memory requirements and argue that implementing a causal memory using the External Pagers is very attractive.

1 Introduction

A distributed shared memory is a memory management system which provides a uniform access interface to local and remote objects. While this functionality is very attractive, its implementation may lead to some efficiency issues. The objective of this paper is to propose an efficient implementation of a causal distributed shared memory based on the Mach memory management facilities [1]. A causal memory may is a memory in which read operations return a consistent value with respect to causally related operations. Two aspects must be considered: the consistency protocol that is used to maintain the memory consistency, and the causal dependencies management.

We concentrate on memories in which objects are replicated among the sites. A mechanism for managing object consistency is required. Two main protocols may be used according to the memory consistence they maintain (i.e: strict or released). Strict consistence [2] may be defined as allowing

multiple readers **or** *one writer* to execute in parallel on an object. This protocol provides the programmer with the vision of a global shared memory but may lead to high message traffic. On the other hand, a released form of consistency which allows *multiple readers* **and** *one writer* to execute in parallel on an object, allows more concurrency and increases the system efficiency. We propose to use this released form of consistency for building our causal memory.

However, the released consistency protocol cannot ensure by itself the property of causality. Some dependency management mechanism should be added to the protocol. The management of dependencies requires that read operations return a causally consistent value. Dependency management is the major issue in implementing a causal memory based on a released consistency protocol.

Causal memories have a high potential for increased performance, and are the subject of current research. Different forms of released consistency has recently been introduced, and the Clouds system has proposed algorithms for implementing a causal memory in which multiple writes are allowed to execute in parallel on copies of a given object [3]. A vector timestamp protocol is used to keep the causality relations. Some other works deal with both released and strict consistency protocols, and leave to the programmer the choice of the protocol [4][5]. Munin [6] requires the user to specify the type of its application, which may be difficult for him.

This work has been done in the framework of the Guide project (Grenoble Universities Integrated Distributed Environment) [7]. Guide is a component of Comandos (COnstruction and MANagement of Distributed Open Systems), a project supported by the Commission of European Communities under the ESPRIT program. The aim of Guide is to explore distributing computing, structured in terms of objects and based on a set of heterogeneous workstations. A first prototype of the Guide system has been implemented on top of Unix, and the second version of Guide has been designed on top of Mach 3.0. A preliminary implementation was developed on top of Mach 2.5 [8].

The rest of this paper is organized as follows. Section 2 defines causal memory. Section 3 gives a brief overview of the Guide object-oriented model. The implementation of a causal memory for Guide on top of Mach is described in Section 4. Finally, Section 5 gives some qualitative and quantitative evaluations of the proposed protocol. We conclude in Section 6.

2 Causal Memory

This Section defines what we call a causal memory. Section 2.1 presents the two major forms of consistency: strict consistency and released consistency, and compares their properties with those of a centralized memory. Section 2.2 describes the major issue in implementing a causal memory, that is the management of causal dependencies.

2.1 Strict and Released Consistency

2.1.1 The Strict Consistency Protocol

A consistency protocol is usually based on some readers/writers algorithm which operates on the copies of an object(1). The readers and writers are the sites where the object is accessed (i.e we do not care about individual processes on each site). The strict consistency protocol is based on a form of readers/writers algorithm which follows the rule: **multiple readers or one writer**. Readers are not

(1) Throughout the paper, we use indifferently the term object or page as a set of data which represents the unit of data transfer between pagers and sites.

allowed to execute in parallel with a writer on an object, and writers are exclusive[2]. With such a protocol, a distributed shared memory simulates a centralized one. Indeed, the main property of a centralized memory is that all operations are serialized by the hardware control mechanism. Only one operation is executed at a given time. This property can be kept in a distributed memory, but in this case all the performance expected from distribution is lost. Nevertheless, the absence of a global order on all operation is masked by the strict consistency protocol. Indeed, while there may be several operations in parallel (one per site), the result of any execution is as if all operations were serialized. This is ensured by the property: a read operation on an object returns the value assigned by the most recent write to the object. However, it appears that the strict consistency protocol ensures more consistency than generally needed in distributed environment where the notion of the "most recent write" is not well defined [9]. So the released forms of consistency which allows to have better performance by reducing the level of consistency appear as an interesting research topic.

2.1.2 The Released Consistency Protocol

There are different forms of released consistency. We are interested in those forms which provide a global order on write operations on an object (write operations are not allowed to overlap). Moreover, we want to respect some order on read operations: let us define H as the history of a variable V, $H = (v_1,..,v_n)$. Successive reads yield values with non-decreasing indices in the history. We can summarize the idea of our released consistency by saying that *the effect of a write operation may be delayed.*

The protocol is implemented as a readers/writers algorithm in which readers and one writer are allowed to execute in parallel on different copies of the same object, i.e **multiple readers and one writer**. With such a protocol, two aspects must be considered:

1) Such a memory enforces less consistency than a centralized memory. The property of serialization is not respected (as opposed to the strict consistency protocol). This means that the following situation may arise :

Example (from [9])

Consider two sequential processes P1 and P2 :

P1: Write (O1, 1) ; Read (O2, 0)[3]

P2: Write (O2, 1) ; Read (O1, 0)

Such a situation cannot arise if read and write operations are serialized (at least one of the read operations should return the value 1). Nevertheless, we believe that the absence of serialization has little consequence for a Guide user; in most cases, this will be transparent to him because he does not need to assume the serialization property for implement any algorithm. Indeed, the serialization is often required for low level synchronization algorithms (such as the Dekker Entry Protocol). The Guide user does not need to use such algorithms since high level synchronization facilities are provided to him by the Guide system. Anyway, if the user does assume such a property, the memory management facilities of Mach still allow the use of the strict consistency protocol operating upon specified objects.

---

(2) The netmemory server provided by Mach implements this type of consistency protocol [1].

(3) *read (O, v)* represents a read operation on the object O which returns the value v. In the same way, *write (O, v)* is a write operation which assigns the value v to the object O.

2) Such a protocol does not ensure the causality property. Indeed, we want to insulate the user from the memory implementation issues by providing him a memory which takes into account causal dependencies[4]. The readers/writers algorithm takes care of operations on a single object, and does not consider any causal relations between operations on different objects. Let us take an example to illustrate one of these causal dependencies.

Example (from [3])

Consider two objects O_1 and O_2 that have 0 for initial value.
Consider two sequential processes P_1 and P_2:
P_1: write (O_1,1) ; write (O_2, 1)
P_2: read(O_2 , 1) ; read (O_1 , 0)

As P_2 reads the value 1 for O_2, it means that the operation *write (O_2,1)* has been executed **before** operation *read (O_2,1)*. Thus, *write (O_1,1)* has also been executed before *read (O_1,0)*, and P_2 should have read the value 1 for O_1. The objective of the dependency management is therefore to take into account the relation *before*. The issue of causal dependencies is explained and formalized in the next Section.

2.1.3 Causal Dependencies

The management of causal dependencies is the most important issue in implementing a causal memory. Let op_i and op_j be two operations related by a causal dependency. We write (op_i -> op_j) if op_i must be executed **before** op_j (in other words, op_j may start only when op_i has finished). We define the dependency relations by three properties:

. Property 1: Let P be a sequential process. P= (op_1 ; .. ; op_n) then (op_1 -> op_2 -> ... -> op_n)

. Property 2: If (op_i -> op_j) and (op_j -> op_k) then (op_i -> op_k)

. Property 3: Let P and P' be two sequential processes.
P = (op_1 ; .. ; op_n) and P' = (op'_1 ; .. ; op'_m).
then (op_i -> op'_j) if

. op_i is a write operation which assigns value v to the variable V at date d, and

. op'_j is a read operation on the variable V, which returns the value v at date d', $d<d'$.

The dependencies induced by Property 1 correspond to the sequential aspect of a process. We call these dependencies **intra-process dependencies**. In the same way, the dependencies induced by Property 3 correspond to inter-process communication and are called **inter-process dependencies**.

In the previous example, we have the following dependencies: (write(O_1,1) -> write(O_2,1)) and (read(O_2,1) -> read(O_1,0)) by Property 1 ; (write(O_2,1) -> read(O_2,1)) by property 3. Then (write(O_1,1) -> read(O_1,0)) is generated by Property 2 and implies that the read operation on O_1 returns the value 1 instead of the value 0.

2.1.4 Summary

A *causal memory* is a memory which preserves the causal dependencies between operations (as defined in 2.1.3). The operations executed by remote processes sharing data are partially ordered.

(4) Demonstrating that causal memories are appropriate for programmers is out of scope of this paper, however many papers have shown this appropriateness [3] [9].

A causal memory manages this partial order. Thus, causal memories are useful for distributed applications in which communication between remote processes is achieved through shared data. The proposed solution is based on a released consistency protocol, completed by the management of both forms of causal dependencies (i.e intra-process dependency and inter-process dependency). Our released consistency protocol does not allow multiple writers to execute in parallel on an object. This exclusion of writers is in fact not required for implementing a causal memory. However, allowing multiple writers to execute in parallel would require the management of the versions of objects. Because the causal memory is implemented in the pagers, exclusive writers allows to implement a causal memory in a very simple and efficient manner (without managing versions of objects). Thus, the released consistency protocol that is used only allows multiple readers and one writer to execute in parallel on an object.

3 Overview of the Guide Object-Oriented Model

While the proposed memory is somehow independent from the Guide system which is implemented on top of Mach, we describe the main feature of the Guide model in order (1) to situate the context of the work and (2) to prove the correctness of the proposed implementation. The Guide model includes the followings aspects :

- An **object** model: objects provide a convenient means for application structuring. Objects are persistent, i.e. their lifetime is not related to that of the execution unit in which they were created. Within the system, they are designated by low-level, location-independent identifiers. The object model is accessible to the users through a specific language whose run-time system is implemented on top of the Guide kernel.
- A computational model, which provides the user with the concept of a **job**. A job represents a virtual machine, which hides the details of distribution, and provides mechanisms for concurrency control. A job groups together in a common address space a set of objects and a set of threads of control, called **activities,** that operate upon these objects. Communication between activities belonging or not to the same job is achieved through **shared objects**. For example, in figure 1, the two jobs *Job1* and *Job2* share the objects *O3* and *O4*.

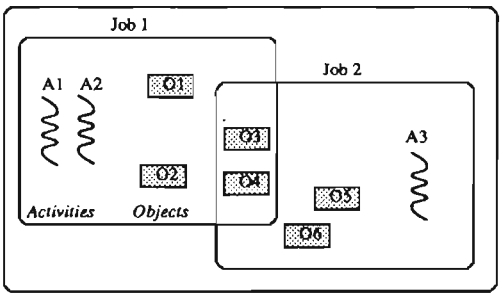


Figure 1: sharing of objects

Guide also provides the **distribution of jobs**: jobs may be distributed on a set of sites. A diffusion mechanism allows a job to dynamically extend to remote sites. Thus, a job may be seen as a set of representatives, also called local jobs, one on each site to which it has diffused. Figure 2 illustrates distribution for two jobs. Job1 is entirely represented on site Node1, whereas Job2 is composed of two local jobs Job 2/Node1 and Job 2/Node2.

Thus, the main notions that must be considered for our proposal deal with the concepts of jobs, activities, shared objects and distribution of jobs.

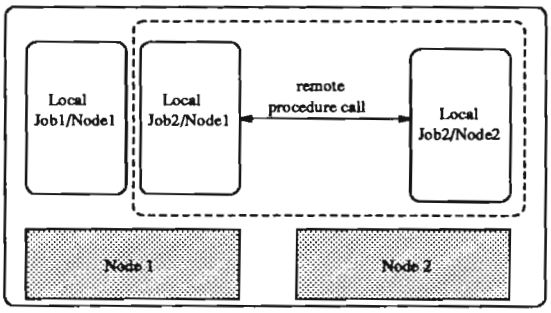


Figure 2: distribution of jobs

4 Causal Memory for Guide on Top of Mach

The objective of this Section is to describe an implementation of a causal memory for Guide on top of Mach. Section 4.1 recalls the principles of the Mach memory management facilities and Section 4.2 proposes a realization of our causal memory on top of Mach.

4.1 Memory Management Facilities on Top of Mach

The Mach kernel allows the user to provide paging services outside the kernel. This is based on external servers (or pagers). A pager allows to create memory objects (i.e chunks of memory), identified by ports[5]. To address a memory object, a thread maps it in the virtual address space of its task. Once an object is mapped, page faults on this object are processed as usual page faults. The kernel sends a page fault to the port which identifies the object. This allows to implement two main architectures of pagers:

- A centralized architecture, in which an object is managed by a unique server. Paging may be remote.
- A cooperative architecture, in which pagers cooperate to maintain object consistency. Paging is always local: a kernel always sends a page fault to its local pager.

Figure 3 illustrates these two architectures. We will see in the next Sections that the cooperative architecture is more adequate than the centralized one for implementing our causal memory.

4.2 Proposal for a Causal Memory for Guide

4.2.1 Guide on Top of Mach 2.5

Let us briefly describe the current implementation of Guide on top of Mach in order to specify the interaction between the Guide concepts and the pagers. Mach provides the concept of a task which is a multi-threaded address space. Thus, each *local job* is implemented by a task and each *local activity* is implemented by a thread. Object sharing between activities in the same local job is implicit because the threads implementing these activities share the same address space. Object sharing between activities in different local jobs is implemented using pagers which allow object sharing between different tasks on the same sites or on remote sites. Before invoking a Guide object, a local

(5) A port identifies a unique Mach object in the network.

(6) Throughout the paper, we use indifferently the term object or page as a set of data which represents the unit of data transfer between pagers and sites.

job must bind the object in its context. This binding operation maps the object into the address space of the task which represents the local job.

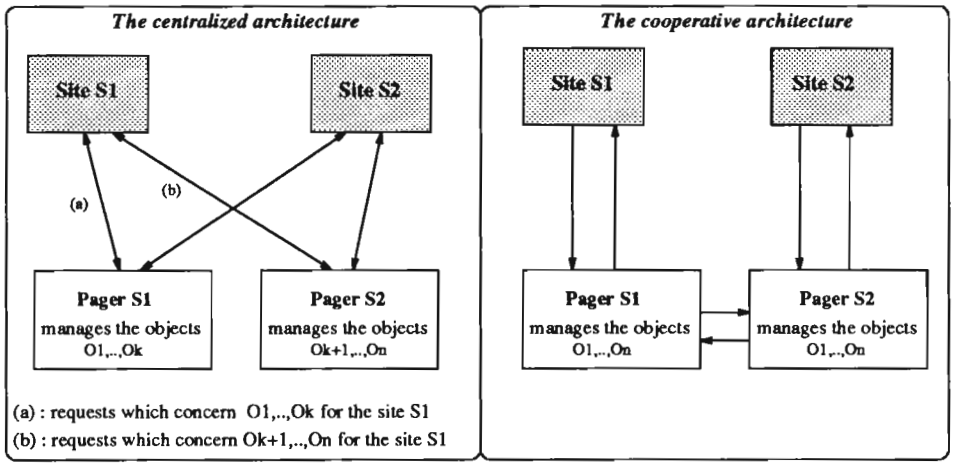


Figure 3: Centralized and cooperative organization of pagers

4.2.2 Proposal

Our proposal consists of implementing a causal memory based on a released consistency protocol. The released consistency protocol is directly enforced by pagers. Pagers apply the rule *one writer and multiple readers* on objects (i.e readers are not invalidated when there is a writer). As a consequence, readers are not always reading the last version of objects. The management of dependencies is therefore in charge of specifying the latest moment when readers should be invalidated so as to keep causal dependencies valid.

While the dependency management can be easily implemented in the object-invocation mechanism [3], we have choosen to implement it in pagers. Indeed, implementing efficiently the dependency management in the object-invocation mechanism requires the knowledge of the behavior of any method, in the sense that we must be able to determine whether or not a method modifies the state of an object. Such a criterion may lead to consider any method which potentially may write as a writing method. The expected gain is consequently reduced since the efficiency of shared memory relies on the fact that the number of read method is significant. Moreover, bracketing method invocations with calls to the consistency protocol is cumbersome, especially if methods have short execution times.

Thus, as pagers have the information which allows them to determine the types of potential accesses (read/write) to each object, we propose to use this information to manage the causal dependencies. This provides the additional advantage of implementing a stand alone causal memory. The dependency management is explained below.

4.2.3 Management of the Causal Dependencies

For more simplicity, we consider, in a first step, that there is a unique pager in the network. Distributed pagers are considered in Section 4.2.4. Pagers have to manage both intra-process and inter-process dependencies, as defined in Section 2.1.3. In the context of Guide, the management of dependencies requires the following properties:

Inter-process dependencies requirement: if an activity invokes successively two shared objects O1 and O2, it will work on consistent copies of O1 and O2. In other words, if O2 has been

assigned a value vO2 before O1 has been assigned a value vO1, then if the activity accesses the value vO1 for O1, it must accesses a value for O2 at least as recent as vO2.

Intra-process dependencies requirement: if an activity successively invokes the same object O, then the second invocation will work on a copy of O which is at least as recent as the copy of O used by the first invocation.

In fact, the pager does not know when objects are invocated, and by which activity they are. Thus, the pager cannot operate for the dependencies management at the same time an object is invocated. It can however operate when providing a copy of an object in response to an object fault. The two next Sections explain the idea.

A) Managing the Inter-process Dependencies

Let us define states of an object copy mapped on a given site: a copy of an object is in the **old** state when it does not represent the current version of the object. Then, a very simple solution for satisfying the inter-process dependency requirement (as defined previously) consists in **invalidating all objects which are in the old state and cached on a site S before sending to S (in response to an object fault) an up-to-date version of an object which was previously in the old state.** There may be fault requesting an object which was previously in the *old* state either when the *old* object has been invalidated, or when the involved kernel requests write rights on the object (indeed, an object in the *old* state can only be accessed in read mode). As only the faults requesting an object which was previously in the *old* state require some invalidations of other *old* objects, the faults requesting an object which is not shared between sites do not involve invalidations (since only the objects that are shared may become *old*). Notice that there can be no more invalidations than with the usual strict consistency protocol since, as soon as an object becomes *old*, some invalidations are avoided.

So, when sending an up-to-date version of an *old* object O on a site S, the pager previously invalidates all copies of shared objects which are not as recent as the copy of O which will be sent. This ensure that an activity cannot access successively two copies of objects O1 and O2 such as the accessed copy of O2 is older compared to the accessed copy of O1. While this solution is convenient due to its simplicity, we propose to use the Mach concept of task to improve it by reducing the number of invalidations of *old* objects. Indeed, the pager knows, for each object, in which tasks it is mapped. Let us define a relation *Shares (T, T', S)* which related two tasks *T* and *T'* which are present on a site *S*. The relation *Shares* is defined as follow:

- If T and T' map a common object on the site S, then Shares(T, T', S).
- If Shares (T, T', S) and Shares (T', T'', S) then Shares (T, T'', S).
- If Shares (T, T', S) then Shares (T', T, S).

The relation *Shares* defines a set of tasks which are related through shared objects (in a direct or indirect manner).Then, **when the pager sends an up-to-date version of an object O in response to an object fault, the set of old objects which have to be invalidated is reduced to old objects which (1) are mapped in tasks which map O or (2) which are in relation Shares with tasks which map O.** The following example illustrates this idea.

Example

T1 is a task on site S, and T2, T3, T4 are three tasks on S'. Objects O1,..,O4 have 0 for initial value.

Suppose the following execution:

Task T1, Site S: **write (O1, 1)** *; write (O2, 1)* .....*O1 and O2 becomes old on S'*

Task T2, Site S' : *read (O2, 1) ; write (O3, 1)*
Task T3, Site S' : *read (O3, 1) ; write (O4, 1)*
Task T4, Site S' : *read (O4, 1) ;* **read (O1, 1)**

In fact, T4 could not read another value than 1 for O1 since there are the relations (write(O1, 1) -> write(O2, 1) -> read(O2, 1)-> ... -> read(O1, 1)). But, if T2 reads the value 1 for O2, it means that an up-to-date version of O2 has been sent to S'. Thus, every old objects on S' which may be accessed by tasks which are in relation Shares (that is T3 and T4) with the tasks which have mapped O2 (that is T2) have been invalidated before the sending of the up-to-date version of O2. Therefore, O1 has been invalidated and the copy accessed by T4 is at least as recent as the accessed copy of O2 by T2.

Thus, we propose two solutions for inter-process dependencies management: a simple one which invalidates all *old* objects of a site S before sending an up-to-date version of an object to S, and a more complex one which allows to reduce the number of *old* object invalidations by using the concept of task. However, performance of this second solution will have to be compared with the simplest one. Indeed, the cost of the use of information about tasks (i.e the management of the relation *Shares*) may be too expensive, leading to adopt the previous and simplest solution.

B) Managing the Intra-process Dependencies

If the management of intra-process dependencies is straightforward when processes do not migrate (or diffuse) to several sites, it leads to some difficulties when a process is allowed to execute on a set of sites. Process migration is very useful for fault-tolerance or load-balancing. Assuming that a process executes on only one site would be too restrictive. Thus, we have choosen to take into account the case of processes which are allowed to migrate, in such a way that it does not cost when they do not migrate. In Guide, because of the sequential character of an activity, we must satisfy the following property, as illustrated in the next example:

Example

Activity A1 on site S1:
Invocation(O)
A1 diffuse on S2 .....> Activity A1 on site S2 :
Invocation(O)

Due to the sequential aspect of an activity, the copy of O accessed on site S2 must be at least as recent as the copy of O accessed on S1. Thus, an inconsistency may arrive if the copy of O on site S1 represents the up-to-date copy of O, and the copy of O on S2 is an old copy of O.

To solve this problem, the intra-process dependencies are managed at the instant an activity migrates from one site to another. The principle is the following: **when an activity A migrates from site S to site S', let T be the task of the thread which represents A on S', then the pager invalidates either every object O which is in the old state on S' (this is the simple solution), or each object O which is in the old state on S' and which is mapped in T or which is in relation Shares with T (this is the more complex solution which allows to reduce the number of invalidations as defined previously in 4.2.3.A).** The reader may think that this solution is not convenient since each diffusion requires the emission of a synchronous message to the pager. Nevertheless, if some objects must be invalidated, then the cost of the message is balanced by the fact that the invalidations of such objects have been delayed to the latest moment. Moreover, we will see in the next Section that this communication

between the activity and the pager is local. This allows to locally share information between the activities and the pager, so that an activity sends a message to the pager only when there are objects to invalidate. Note that we do not need to synchronize the access to this information since it is only modified by the pager.

4.2.4 Taking Distribution of Pagers into Account

Because of the distribution of pagers, the required information (like the list of *old* objects on a given site) is distributed. In Section 4.1, we have presented two possible architectures for the pagers: a centralized one and a cooperative one. While the current implementation of the Guide prototype uses the centralized architecture, we intend to switch to a cooperative architecture for implementing the proposed causal memory. Indeed, there are two advantages of this decision:

- First, it allows to have local paging, instead of remote paging (as with the centralized architecture).
- Second, with a cooperative architecture, information is distributed in an adequate manner for the protocol requirements. Indeed, each pager knows every thing about the objects that are cached into the site it manages (like the mapping of objects to the tasks of its site). This information is sufficient to manage the dependencies.

Pagers have to cooperate in order to ensure that there is no more than one writer on a given object at a specified time. For this, we associate an *owner pager* to each object. The owner pager of an object is in fact the pager of the site which has the latest version of the object. While any kernel sends an object fault to its local pager, the receiving pager may forward the request to the owner pager of the given object. If the owner migrates from one site to another, a forward link allows to find it.

Finally, the overall execution is described as following. An object which is mapped on a site can have four states as seen by the local pager:

- **read**: the site has *read* rights on the object, and has the latest version of the object.
- **write**: the site has *write* rights on the object, and has the latest version of the object.
- **old**: the site has *read* rights on the object, and has an old version of the object.
- **invalid** the object has been invalidated, which means that it is conceptually not more present on the site.

In addition, a pager manages a set of attributes for each object:

- **owner**: gives the identification of the owner pager.
- **page_out**: true if the object has been paged out from the local kernel. When receiving a page fault request from the local kernel, this information allows to know whether the page fault has been caused because the accessed page was paged out. In this case, the pager has simply to provide the kernel with the paged out copy of the page.
- **readers_list**: lists the pagers of the sites which have the object in reading mode. This allows the owner of a page to notify all readers when the page is accessed in write mode.
- **tasks_list** : lists the tasks of the site which map the object.

For each task which is represented on the current site, the pager manages a relation **Shares**. *Shares (T)* gives the list of the tasks which are related with task T through shared objects (as defined previously). The relations *tasks_list* and *Shares* allows, when sending an update version of a shared object to the local kernel, to determinate the tasks for which *old* objects must be invalidated. As previously said, a more simple solution allows to avoid the management of the relation *Shares* but

leads to more invalidations: every *old* objects that are on the local site are invalidated when sending an up-to-date version of a shared object to the local kernel.

Because the entire protocol is difficult to describe, we have choosen to illustrate the protocol by simplified pseudo-algorithms. Detailed algorithms are given in Annexe A.

1) When receiving an object fault for an object O:

If O has just been paged out, and is not is the invalid state, the pager provides the requesting kernel with its local copy of O and returns.

If the pager is not the owner of O, it communicates with the pager owner of O in order to get an up to date copy of O. The owner of O may change regarding to the requesting mode for O:

If O is requested in write mode then the local pager becomes the owner of O and O turns in the old state with read rights only on the other sites where it is mapped.

Then, before sending the copy of O, the pager invalidates:

- *with the simple solution: every old object on the current site*
- *with the more complex solution: the old objects on the current site which (1) are mapped in tasks which map O or (2) which are mapped in tasks in relation Shares with the tasks in which O is mapped.*

2) when receiving a notification of an activity migration:

Let T be the task of the thread which represents the activity on the destination site.

The pager of the destination site invalidates:

- *with the simple solution: every old objects on the local site*
- *with the more complex solution: the old objects which are mapped in T, or which are mapped in tasks which are in relation Shares with T.*

5 Evaluation

5.1 Adequacy of Mach for the Protocol

We can make the following remarks regarding the adequacy of Mach for the proposed protocol:

First, due to the fact that Mach allows the user to provide paging services outside the kernel, we had two possibilities: implementing our causal memory in the object-invocation mechanism or in the pagers. We have shown that the possibility of managing dependencies in the pagers is very attractive. Indeed, it avoids bracketing method invocations with calls to the consistency protocol without having an efficiency decrease.

Secondly, the External Pager facilities also allow to implement a specific architecture of pagers. We have shown that the cooperative architecture completely satisfies our causal memory requirements. Moreover, it allows local paging. In addition, since each object can be treated in an independent manner, it allows to use both a strict consistency protocol for specified objects (which require the serialization property for example), and a released consistency protocol for other objects.

Finally, the invalidation mechanism avoids to refresh copies of objects which are no more accessed. Nevertheless, the kernel paging interface does not allow to request more than one object invalidation per message. As the proposed protocol often requires the invalidation of a set of objects

at the same time, the ability for a pager to request more than one object invalidation in a message would be useful for our protocol.

5.2 Some Quantitative Evaluations

The following figures compare the cost of the released consistency protocol implemented by the cooperative architecture of pagers with the usual protocols, which are strict consistency protocols implemented either by the centralized or by the cooperative architecture of pagers. Comparisons are given in terms of the number of exchanged messages.

Figure 5 (on next page) presents an evaluation in a favorable case, where a page is both accessed in the write mode on site S1, and in the read mode on another site S2. Recall that, with the strict consistency protocol, there cannot be parallel read and write accesses on the given page on sites S1 and S2. Thus, we suppose that, in the case of the strict consistency protocol, each of these accesses causes a page fault.

In fact, consequences due to the dependency management are not represented in this evaluation. Remember that the effects of the dependency management is to invalidate pages in order to prevent inconsistencies. Then, even if the dependency management leads to an invalidation of the accessed page on the site S2 (which has an old version of the page since the page is on S1 in write mode) at the time of the S2's second access (namely, *d)read*), the proposed protocol is still more efficient than the strict consistency protocol (see the case n=2).

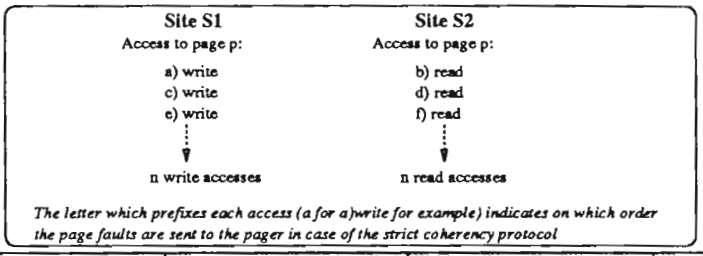
Figure 6 presents an evaluation in an unfavorable case, where a page is both accessed in the write mode on site S1 and on site S2.

The reader can see that the cost of the proposed protocol is in fact the same as the cost of the strict consistency protocol implemented by the cooperative architecture of pagers. When compared with the centralized architecture of pagers, the cost of the proposed protocol seems to be acceptable: if the number of messages may be greater, the number of remote communications is smaller.

5.3 Evaluation Conclusion

As the proposed protocol has not been yet implemented, we do not give measurements in this paper. Nevertheless, we believe that the previous evaluations show the gain that we could expect from such a causal memory because they reflect the real number of exchanged messages.

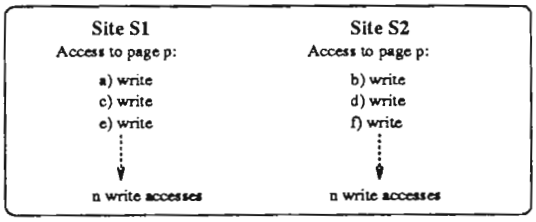
Anyway, we can remark that the gain is due to the delayed of invalidations of pages, which leads to both a **decrease of the number of page transfers and a decrease of the number of invalidations** compared with a strict consistency protocol. To be fair, it should be stated that, although the number of transfers and invalidations is reduced, there is more time spent to manipulate the data structures like *tasks_list* , or others. But this CPU time is negligible compared to the gain we get when reducing the number of remote communications. Moreover, we proposed two solutions for the dependency management (see Section 4.2.3): a very simple one which implies more invalidations than the second, but which is still more efficient than the usual strict consistency protocol. The second solution is more difficult to evaluate: it requires less invalidations but may lead to the management of somewhat complex data structures.



The letter which prefixes each access (a for a)write for example) indicates on which order the page faults are sent to the pager in case of the strict coherency protocol

	Strict consistency + Centralized arch. of pagers	Strict consistency + Cooperative arch. of pagers	Released consistency + Cooperative arch. of pagers
Number of page faults	2n	2n	n
Number of messages	8n−2	12n − 4	12
Number of remote messages	4n−2 to 8n−2	4n − 2	4
Number of page transfers	3n	3n + 1	4
Numb. of remote page transfers	n to 3n	n	1
Case n = 2			
Number of page faults	4	4	2
Number of messages	14	20	12
Number of remote messages	6 to 14	6	4
Number of page transfers	6	7	4
Numb. of remote page transfersss	2 to 6	2	1

Figure 5: evaluation in a favorable case



	Strict Consistency + Centralized arch. of pagers	Strict Consistency + Cooperative arch. of pagers	Released consistency + Cooperative arch. of pagers
Number of page faults	2n	2n	2n
Number of messages	8n−2	12n−4	12n−4
Number of remote messages	4n−2 to 8n−2	4n−2	4n−2
Number of page transfers	4n−1	6n−2	6n−2
Numb. of remote page transfers	2n−1 to 4n−1	2n−1	2n−1
Case n=2			
Number of page faults	4	4	4
Number of messages	14	20	20
Number of remote messages	6 to 14	6	6
Number of page transfers	7	10	10
Numb. of remote page transfers	3 to 7	3	3

Figure 6: evaluation in a unfavorable case

6 Conclusion

In this paper, we have described a simple mechanism for implementing an efficient distributed shared memory for the Guide object-oriented distributed system on top of the Mach kernel. The proposed memory is called a causal memory in the sense that it respects the causal dependencies between operations. Such a memory does not simulate a centralized one because it does not provide the serialization property. Nevertheless, the serialization property is not required by most applications, and we believe that it will be sufficient for the Guide users. Moreover, the proposed mechanism can be used in parallel with the usual strict consistency protocol. This allows to use a strict consistency scheme only when needed.

Our proposal is innovative since there are few systems which implement such a memory. Causal memories just start to become a subject of investigation, and most of the relevant papers only present a theoretical point of view. Thus, we believe that our proposal is interesting because we describe a simple and efficient implementation of such a memory by using the Mach memory facilities. The mechanism seems to be efficient, since the number of communications (and especially the number of page faults) is reduced compared to the usual protocol for the consistency management (based on a strict consistency scheme). Finally, as our proposal is based on the External Pagers facilities provided by Mach, we believe that the proposed memory could be used by any system which is implemented on top of Mach.

Acknowledgements

I would like to thank Professor Sacha Krakowiak and Xavier Rousset de Pina for their help in reviewing this paper.

The Guide project is supported by the commission of European Communities through the ESPRIT program in project COMANDOS (Construction and Management of Distributed Open System), the Universities of Grenoble (Institut National Polytechnique de Grenoble - Université Joseph Fourier) and Centre National de la Recherche Scientifique.

Annex A
Algorithms

```
PROC Pager_receives_request_for_page_from_kernel (page, mode)
[ This request is executed by a pager when it receives a request for a page fault from the local kernel]
 if (page_out [page]) and (not state[page]=invalid) then return the page to the requesting kernel
 else if owner [page] = myself
  /* this means that the requested mode is write when the current mode of the page is read */
                for each pager P in readers_list [page]
                do  Pager_ask_for_page_to_become_old (P, page)
                list_readers [page] = Null ; state [page] = write
                return the page to the requesting kernel
          else /* I am not the owner */
             if (state [page] = read )
                     Pager_ask_for_rights_to_owner ( owner [page], page )
                    return the rights to the requesting kernel
              else  Pager_ask_for_page_to_owner (owner [page], page, mode)
                    statee [page] = mode
                     if (state = write ) owner [page] = myself ;  readers_list [page] = NULL
                     /* because the requesting kernel will receive a new version of the page,
                      we must manage the inter-process dependencies */
                   for  each task T in  tasks_list[page]
                     for  each task T' in  Shares[T]
                      for  each page p such as (mode[p] = old) and (T' in tasks_list[T])
                        do state [p] = invalidate ;  invalidate (p)
                   return the page to the requesting kernel
 EndProc

 PROC  Owner__receives_request_for_page_from_pager(page,  mode)
 [ This procedure is executed by a pager which is (or has been) the owner of the  given page. This procedure
replies to the request  Pager_ask_for_page_to_owner, coming from another pager ]
      if (owner [page] <> myself ) forward_to (owner [page] )
      else if (mode = write)
                 owner [page] = asking_pager ; mode [page] = old
                  for each pager P in readers_list[page]
                  do Pager_ask_for_page_to_become_old(P, page)
           else  readers_list [page] += asking_pager ; state [page] = read
                /* in fact, we choose to reduce the rights of the current site, because it is
               more probable that the requesting site will soon access the page in write mode */
          return the page to the requesting pager
 EndProc
```

```
PROC  Owner_receives_request_for_rights_from_pager  (page)
[ This procedure is executed by a pager which is (or has been) the owner of the  given page. This procedure
replies to the request  Pager_ask_for_rights_to_owner, coming from another pager ]
/* as  the requesting rights are necessary in write mode, this procedure executes as the previous procedure,
in the case (mode = write) */
EndProc

PROC  Pager_receives_order_for_page_to_become__old  (page)
[  This  procedure  is  executed  by  a  pager,  when  it  receives  the  request
Pager_ask_for_page_to_become_old  from another pager]
    state [page] = old
EndProc

PROC  Pager_receives_request_for_map_from_task  (task,  page)
[ This procedure is executed by a pager when it receives a request for binding an object  from a local job]
    Shares  [task] += tasks_list  [page]
    tasks_list  [page] += job
    if (state [page] = old )
            state  [page]  =  invalidate
             if (page_out  [page]) page_out  [page] = false
             else invalidate (page)
EndProc

PROC  Pager_receives_notification_of_job_extension(job,  task)
[ This procedure is executed  when an activity migrates from one site to another (assume S' ), and when there
is objects to invalidate. It is the local  pager (on S' ) which executes this procedure]
  for each task T  in  Shares[task]
    for each page P such as (T in tasks_list [P] or ) and (mode [P] = old)
    do  invalidate (P) ; mode [P] = invalidate
EndProc
```

Bibliography

[1] Alessandro Forin, Joseph Barrera, Richard Sanzi. The Shared Memory Server. *Usenix*, winter 1989.

[2] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[3] M. Ahamad, P. W. Hutto, R. John. Implementing and Programming Causal Distributed Shared Memory. *11th International Conference on Distributed Computing Systems*, May 1991.

[4] W. K. Giloi, C. Hastedt, F. Schoen, W. Schroeder-Preikschat. A distributed implementation of shared virtual memory with strong and weak coherence. *Distributed Memory Computing, 2nd European Conference*, EDMCC2, April 1991.

[5] P. K. Sinha, H. Ashihara, K. Shimizu, M. Maeckawa. Flexible User-Definable Coherence Scheme in Distributed Shared Memory of Galaxy. *Distributed Memory Computing, 2nd European Conference*, EDMCC2, April 1991.

[6] Bennett, J. K. , Carter, J. B. and Zwaenepoel, W. . Munin: Distributed Shared Memory based on Type-Specific Memory Coherence. *Proc 2nd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 168-175, 1990.

[7] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme. Architecture and Implementation of Guide, an object-oriented distributed system. *Usenix Computing Systems*, 4(1) Winter 1990.

[8] F. Boyer, J. Cayuela, P.Y. Chevalier, A. Freyssinet, D. Hagimont. Supporting an object-oriented distributed system: experience with Unix, Mach and Chorus. *Proceedings of the 2nd Symposium on Experience with Distributed and Multiprocessor Systems*, Atlanta, Mars 1990.

[9] P. W. Hutto, M. Ahamad. Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. *10th ICDS*, 1990.