# A Distributed Consistency Server for the CHORUS System

*Vadim Abrossimov*
*François Armand*
*Maria Inés Ortega*

Chorus systèmes
6, avenue Gustave Eiffel, F–78182, Saint-Quentin-en-Yvelines (France)
Tel: +33 1 30 64 82 56, Fax: +33 1 30 57 00 66, E-mail: ines@chorus.fr

## 1. Introduction

This paper describes how distributed shared memory is being implemented on the CHORUS® system, by a set of servers running outside the CHORUS Nucleus itself. These servers cooperate to implement, in a decentralized fashion, distributed consistency of data between multiple sites. The algorithms used for that purpose derive from those described in[Li89a] . This service will be used to provide Distributed Shared Memory in the CHORUS/MiX™ V.4 subsystem running on top of the CHORUS Nucleus. The CHORUS/MiX V.4 subsystem is compatible with UNIX® SVR4 and is designed to provide a Single System Image on multicomputer architectures.

The next sections will briefly describe the CHORUS system and the CHORUS/MiX V.4 distributed system, and will detail some of the reasons why Distributed Shared Memory is needed when building up a Single System Image.

We will then spend some time to summarize similar mechanisms designed and implemented in related projects. After having summarized the CHORUS Virtual Memory interface and the main goals of the design of our servers, we will describe their design and some implementation details. We will conclude with some lessons learned and outline future developments of these servers.

## 2. CHORUS/MiX

### 2.1 CHORUS Architecture

A CHORUS System is composed of a small-sized **Nucleus** and of possibly several **System Servers** that cooperate in the context of **subsystems** to provide a coherent set of services and a user interface. A detailed description of the CHORUS system can be found in[Rozi88a] . Among the other systems that have adopted similar architectures one will find: Mach [Acce86a] ,V-system [Cher88a] and Amoeba [Mull87a] are some examples.

In: Proc. of SDMS III, Symposium on Experiences with Distributed and Multiprocessor Systems, Newport Beach, CA, March 26-27, 1992

® CHORUS is a registered trademark of Chorus systèmes.

® UNIX is a registered trademark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

### 2.2 CHORUS Nucleus Abstractions

The **actor** defines an address space that can be either in user space or in supervisor space. In the latter case, the actor has access to the privileged execution mode of the hardware. User actors have a protected address space. One or more **threads** (*light weight processes*) can run simultaneously within an actor. They can communicate using the memory of the actor if they run in the same actor.

Otherwise, they can communicate through the CHORUS IPC that enables them to exchange **messages** through **ports** designated by global unique identifiers (or UI). A message is composed of a (optional) **body** and an (optional) **annex**. Ports may be dynamically inserted into or removed from **port groups**. The CHORUS IPC mechanism allows a message to be sent to all ports of the group (broadcast mode) or to only one port in the group (functional addressing mode).

### 2.3 The CHORUS/MiX V.4 subsystem

MiX V.4 is a CHORUS subsystem providing a UNIX interface that is compatible with UNIX SVR4. It is both BCS and ABI compliant on AT/386 machines. It is composed of a set of cooperating servers that run on top of the CHORUS nucleus and which communicate only by means of the CHORUS IPC. The following servers are the more important:

- The **Process Manager** (PM) provides the UNIX interface to processes. It implements services for process management such as the creation and destruction of processes or the sending of signals. It manages the system context of each process that runs on its site. When the PM is not able to serve a UNIX system call by itself, it calls other servers, as appropriate, using CHORUS IPC.
- The **File Manager** (FM) also refered to as the Object Manager (OM) performs file management services.
- The **Streams Manager** (StM) manages all stream files such aspipes, network access, tty’s, named pipes.

CHORUS/MiX V.4 has been designed to be distributed over a set of sites. The ultimate goal of this design is to provide Single System Image semantics, masking multicomputer topologies to user process. Among the features that must be provided to achieve such a goal, one will find the following: single file name space, transparent access to any file from any node of the multicomputer, unique process identifiers name space, remote execution and process migration capabilities...

Some of these features have already been developed and proven within the previous version of the MiX subsystem (MiX V.3.2) which is compatible with UNIX SVR3.2 systems. The Locus[Pope85a] system although not based on a microkernel has also demonstrated such capabilities.

## 3. Needs for Distributed Shared Memory

Among the reasons to provide distributed shared memory in a UNIX SVR4 compatible distributed system, one may list the following: processes running on different sites may communicate through System V IPC shared memory mechanism, regular files being accessed concurrently from different sites need to be maintained consistent.

Unix systems permit users to map files into their process address spaces. This mapping of files can be shared between several processes. When a file is mapped it can either be read or written by simply reading or writing an address location within the process address space corresponding to the offset of the byte in the file. If the mapping is shared between two processes, every modification made by any of the two processes is immediately visible to the other one. This behavior must be maintained within the distributed system even if the two processes are running

on different nodes.
In a distributed system, in order to achieve good performance when accessing remote files, it is desirable to cache, on the client side, parts of the files being accessed. This raises the problem of maintaining coherence between all these client caches. In CHORUS/MiX this problem is identical to the "mapped file" issue.

In addition, shared memory as opposed to IPC mechanisms provides a simpler abstraction to the application programmer to enable multiple processes to communicate. Thus it seems a quite desirable feature although not without potential pitfalls when misused.

## 4. Related Works

There are numerous works and projects which deal with distributed shared memory consistency.There is no room here to describe all of them. Generic surveys of literature dealing with this topic may be found in[Hell90a] ,[Nitz91a] ,[Stum90a] and[Tam90a] . We will focus on the work conducted in IVY[Li89a] and Leases[Gray89a] .

### 4.1 IVY

IVY is a study of the memory coherence problem in designing and implementing a shared virtual memory on loosely coupled multiprocessors. Shared data is paged between processors.

The protocols assume that every page is owned by a site. The ownership can be fixed or dynamic. In the fixed approach a page is always owned by the same processor. In the dynamic case the owner of a page can change but there is always only one owner at any given time, namely the last site that has had write access to the page. Page owner information is managed according to one of the following strategies: *centralised* where only one site knows all page-site mapping and, *distributed* where several sites cooperate to manage the page ownership.

The distributed policy although more complex to implement provides better throughput. The work descibed in this paper derives from this mechanism, but provides some extensions which will be described later.

IVY’s solutions operate only with access on one page a time. If the Chorus mapper wants to serve many Unix servers it must make sure that the access to the object’s fragments is atomic. Moreover, we support different page sizes; the granularity is not fixed. In a CHORUS system, several algorithms implementing different coherence semantics may coexist: coherence algorithms are implemented by an independant actor outside of the nucleus.

### 4.2 Leases

Leases proposes an efficient fault-tolerant mechanism for distributed virtual memory consistency that handles host and communication failures using physical clocks. A *lease* is a contract that gives its holder specified rights over property for a finite period of time. In the context of caching, a lease grants to its holder control over writes to the covered datum during the term of the lease. The server must obtain the approval of the leaseholder before the datum may be written.

One of the problems of this algorithm is to determine the duration of the lease. It is based on a trade-off between minimizing lease extension overhead versus minimizing false sharing. Short-term leases have a number of significant advantages over longer leases, including lower write delays resulting from client crashes, lower recovery delay from server crashes and reduced false sharing.

We think that this work would be interesting and could be incorporated in the future with our work because it provides a mechanism to avoid thrashing and fault-tolerant problems.

### 4.3 Miscellaneous

Mirage[Flei89a] provides a consistent distributed shared memory using infinite-term leases. Miragés property permits the readers or the (single) writer of a page uninterrupted access to the page for a fixed period of time, regardless of other processes requesting it.

Munin[Benn90a] is a DSM system proposed and currently being developed at Rice University. Instead of a single memory coherence mechanism for all shared memory, Munin employs several different mechanisms, each appropiate for a different class of shared data object.

Another direction is a DSM accomodating heterogeneity. This is a difficult problem, because at the page level, byte and words are the primitives, not types data objects. Arcadés support of heterogeinity has led naturally to sophisticated and powerful kernel-level support for distributed shared memory using a langage approach[Cohn91a] . This problem has been also dealt with in[Zhou90a] and[Stum90a] .

## 5. CHORUS Virtual Memory

The CHORUS Virtual Memory (VM) guarantees local memory coherence and offers tools to build distributed memory coherence[Abro89a] . In this section we only present the VM interface that is used to build external mappers.

### 5.1 Basic Abstractions

A **segment** is a collection of data with an associated name (eg: a file). This name is a **capability** (segcap) exported by external servers called **mappers** (eg: the MiX File Manager) and is built from a server's port Unique Identifier and a key that is meaningful only within that server. These servers manage the implementation of the segments, as well as their protection and designation. A segment can be either expliclity read and written through the sgRead/sgWrite CHORUS system calls, or mapped in an actor's address space.

The nucleus encapsulates the physical memory, holding portions of the segment in a per segment **local cache** object (see figure 1). A **local cache** object is designated by its capability; the server for local caches is the nucleus. In addition, each local cache contains a log of all pages which have changed relative to the segment. With each page in the cache, there is associated an **access right** that describes the possible operations (e.g read/write) allowed at a given time.

On a site, the same cache object is used for both the mapped and the explicit segment access, thus insuring that any modification done through one interface will be immediately visible through the other. In other words, the unicity of the cache object avoids any double caching issue for a segment. When multiple sites use the same segment, the corresponding cache objects (one per site) will be named with different **local cache capability** (lccap), thus enabling a mapper to distinguish between these different local caches, and to implement a distributed consistency maintenance protocol.

Page faults generated by reading or writing the memory associated with a mapped object will, in turn, produce requests to the **mapper** for the corresponding access right and data from the object. When the nucleus wishes to free modified pages, it sends requests to the mapper to write back the modified data.

A **mapper** exports a simple segment access interface (described in the next section) to the nucleus. Conversely, the VM exports an interface allowing a mapper to control the state of the local cache associated with a segment. Both interfaces rely on the CHORUS IPC mechanism.

### 5.2 The Segment Request Interface

The segment request interface provides a mechanism by which a nucleus can demand and return pages of a segment to a mapper. It also provides a means for determining access rights to parts

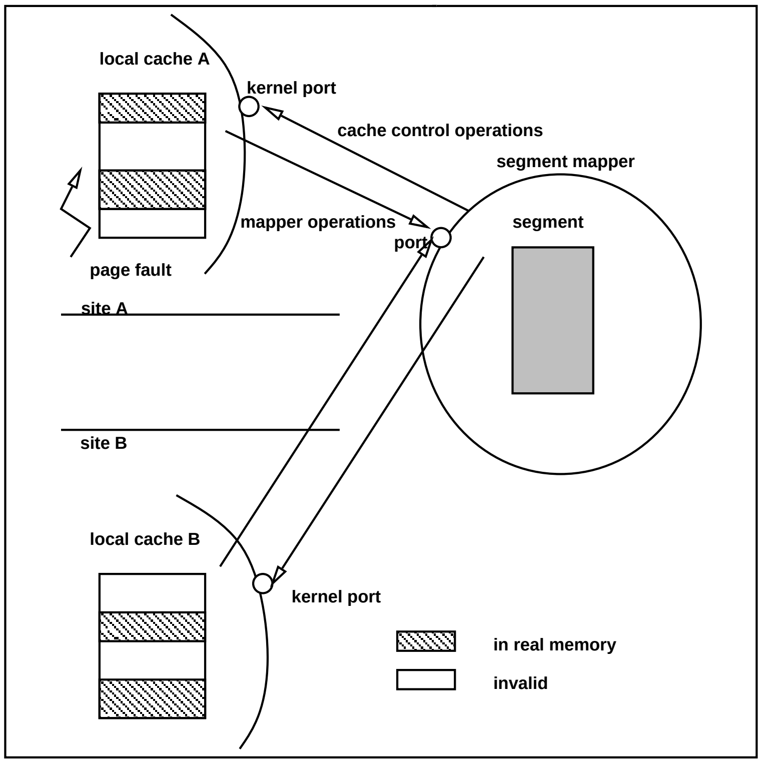

**Figure 1.** – Local Caches

of a segment, without shipping the associated data, and to page out parts of a local cache. The requests which make up this interface are:

- **mpGetAccess** (*segmentCapability, size, offset, requiredAccess:R/W*)
  The **mpGetAccess** request permits a nucleus to request read/write access to a fragment of a segment. If a legitimate access is requested then the mapper will return a success indication.
- **mpPullIn** (*segmentCapability, offset, size, accessRequired:R/W*)
  This request is used by the nucleus for demanding read/write access and the data of the segment to the mapper. If the mapper indicates success then it must supply some data together with the reply. The data returned may already be dirty (not yet saved on backing store). If the mapper indicates failure then the local cache will remain unchanged, and the page fault will not be satisfied. An *mpPullIn* request carries an implicit *mpGetAccess* request for the fragment.
- **mpPushOut** (*segmentCapability, offset, size*) ;
  This request causes data to be updated from a local cache to a mapper. Usually dirty pages are pushed out, but clean pages may also be pushed out upon explicit request of the mapper (see lcFlush below). The mapper can write the data to the segment (eg: on secondary storage) or send the data to another site which has made an *mpPullIn* request.
- **mpCreate** ()
  This request is the means by which the nucleus creates a segment for an internally created local cache (e.g. Swap). In case of success it returns the capability for the segment being

created.

- **mpDestroy** (*segmentCapability*)
  This request permits the nucleus to end the association between a local cache and its segment which was created by an *mpCreate* call.

## 5.3 The Local Cache Request Interface

The local cache request interface provides the means for mappers to maintain control over the data they supply to local caches. We present the **lcFlush** operation which disposes of data in a local cache and **lcSetRights** which changes the access rights associated with a page.

### 5.3.1 lcFlush

**lcFlush** is called by the mapper to invalidate, to change the access rights of, or read a local cache. It takes as arguments an offset, a size, a local cache capability, and a flag. The flag indicates the mode of flush and may be: invalidation mode, invalidation mode and read-cache or change of access from write to read and read-cache. An *lcFlush* operation performed against a local cache may result in one or several *mpPushOut* requests from the CHORUS VM to the mapper. Flushing a local cache is a synchronous operation, and thus will return to the caller when the corresponding *mpPushOut* operations will have completed.

Let us focus on six relevant situations which can occur when a *lcFlush* is called:

1. The flag is *"invalidate"* and the page has not been modified (figure 2.1): in this case the page is invalidated and the operation returns.
2. The flag is *"invalidate"*, the access rights granted to the local cache were "read", but this page has been modified and the modifications are not yet written-back (figure 2.2). This case is possible if an access rights change from write to read has previously occurred. The page is then invalidated and a *mpPushOut()* to the mapper is performed.
3. The flag is *"invalidate"* (figure 2.3), the access rights granted to the local cache were "write", and the page has been modified: the site loses all rights on the page and the page is pushed out to the mapper. In fact, this case is identical to the previous one.
4. The flag is *"read-cache"* and the page has not been modified: an *mpPushOut* to the mapper is peformed (figure 2.4), but the page remains in the site with the same access rights.
5. The flag is *"read-cache | invalidate"*: the page is invalidated and an *mpPushOut* is performed whether the page has been modified or not (figure 2.5).
6. The flag is *"read-cache | change-access"* and the page has been modified: the access rights change from write to read and an *mpPushOut()* occurs (figure 2.6).

### 5.3.2 lcSetRights

The **lcSetRights()** changes the access rights associated with the page from write to read or invalidates the data (recalls all access rights). It receives the offset and size of the fragment being involved, the capability of the segment and an optional invalidate flag.

In the next paragraph we present, two examples of the use of this function made by the mapper.

Figure 3 shows a page with write access right:

1. In the first case (figure 3.1), the page has not been modified so the access right is changed from write to read and the operation returns.
2. In the second example (figure 3.2), the page has been modified, the access right is also modified but the page remains modified with read access rights.

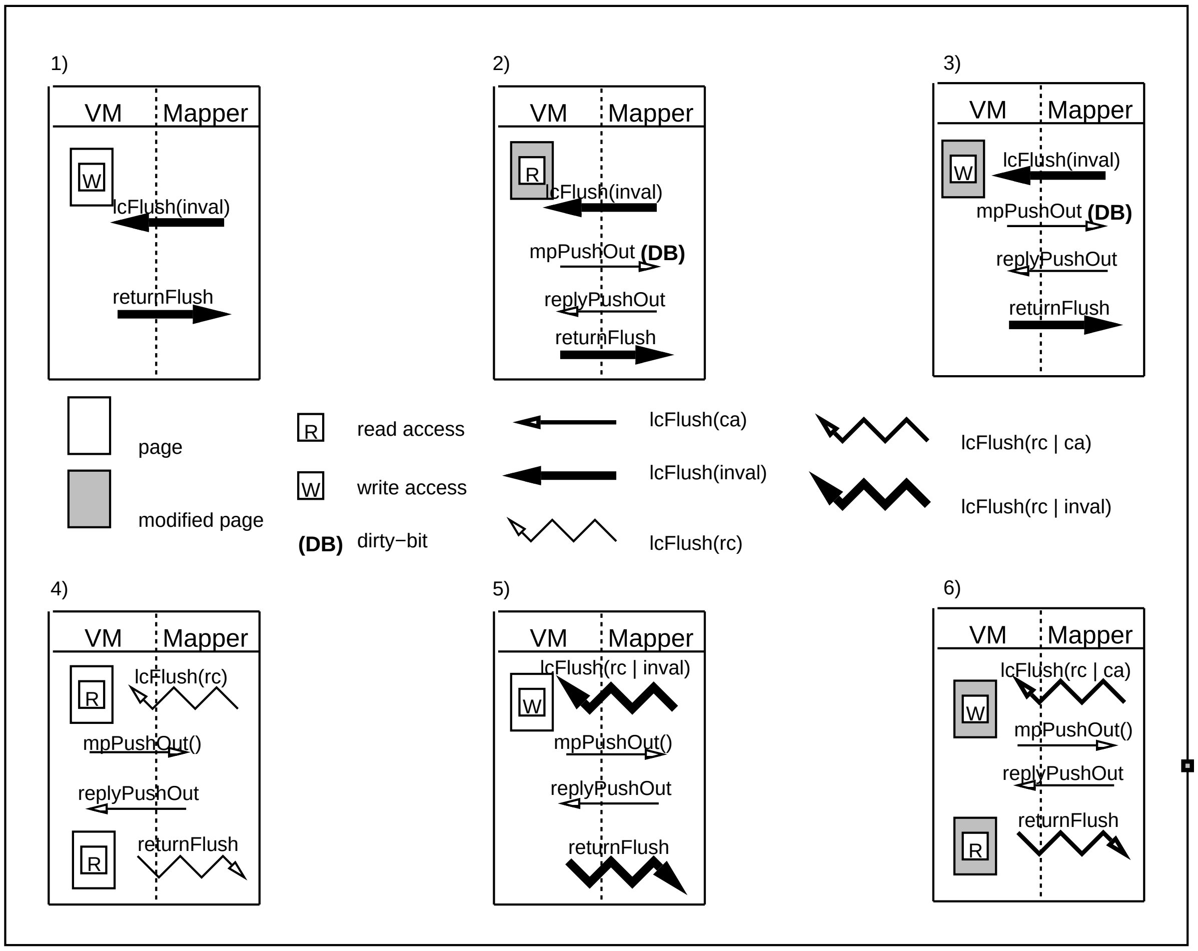

**Figure 2.** – Flush control

### 5.3.3 Data Transfer Mechanism

External Mappers that provide only consistency mechanisms but no storage for segments do not need to access the data of the segment. They just need to move these data around ("receive" and "forward"). The CHORUS segment interface allows one to move data by using a "Data Descriptor" rather than by giving the address and size of the data to be moved. Thus, one can avoid mapping or copying data in a Mapper if it is not needed.

## 6. The Distributed Coherency Server

### 6.1 History and Main Goals

We have already developed a Coherency Server that implements the "centralized" algorithm as described in IVY[Orte91a] . In such a case, when a site A wants to get a page of a segment, it has to send a *mpPullIn* request to the "centralized" mapper. Let us suppose that the requested page was already loaded with "write" access granted on site B. The mapper will then have to reclaim the page from site B (using *lcFlush*) before being able to send this page back to site A.
Such a scheme although simple has the drawback to move the page twice (from site B to the

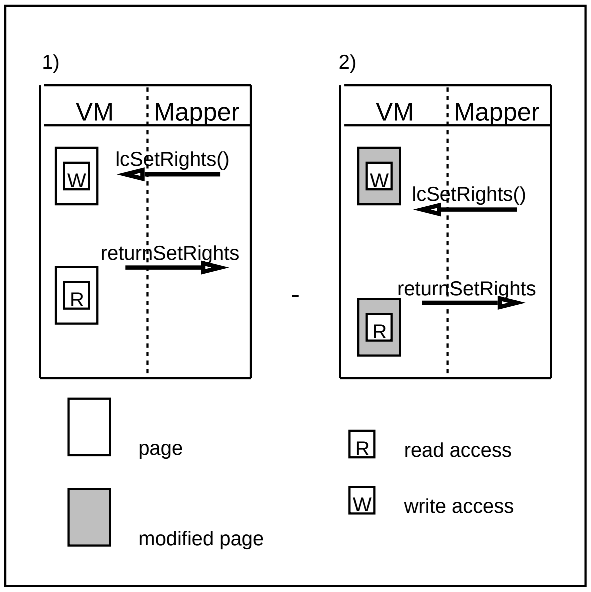

**Figure 3.** – Change access

mapper, and from the mapper to site A). This consumes network bandwidth and makes the mapper a potential bottleneck. Experiments done in IVY have also shown that this algorithm is not the most efficient. Thus, we wanted to implement the "dynamic decentralized" mechanisms as described in IVY.

When designing this service we had several goals in mind:

- Independent Server
  In the same way, mappers are not part of the CHORUS Nucleus, so that system builders may implement the coherency policy they need, we wanted to have consistency be implemented outside any CHORUS/MiX File Manager. This is not only helpful to start coding and debugging, but also provide an easy way to provide distributed consistency for any file system server that exists or will exist. Such an independent server may also be used in any other subsystem that requires the same kind of consistency (eg: Object Oriented subsystems). Due to the encapsulation of the consistency policy, it is also easier to make it evolve without impacting on either the CHORUS VM or the MiX File Manager.

- Support of Unix semantics
  We had some additional requirements: the mapper should at least provide the basic mechanisms to fully support transparent UNIX semantics in a distributed environment. Mainly, one must guarantee that *read*(2) and *write*(2) operations are processed serially. In other words, a read operation occurring concurrently with a write operation on overlapping area of a file, may return data as they were before the write or as they are after the write complete. This must be guaranteed even though the read and write operations are done on large fragments of the file (i.e.: larger than a page). This is an extension to the service provided in IVY which deals only with page access concurrency.

- Heterogeneous Granularity
  In a distributed environment one may have machines running with various virtual page size

(whether they have or not the same instruction sets is another issue): Sun3 machines used 8Kbytes pages whereas other 68K based machines used 4Kbytes pages. We must support such environments. Object Oriented subsystems may also have requirements for consistency that are different from page aligned common needs. Thus, each segment managed by our service has an associated "granularity" attribute, that will help to deal with such requests: a granule is the smallest size of the segment that can be granted to a mapper's client.

### 6.2 Single Writer / Multiple Readers

One of the ways to insure consistency is to block any read operation on a fragment of a segment while a write operation is in progress on the same fragment. When a write operation has completed, one can unblock the pending read operations after having given them back the new image of the fragment as modified by the write. Reciprocally when a write operation starts it is blocked until all the "in progress" read operation have completed. Thus, at any given time, one may have over the network either **several read-only copies** of the fragment or **one, and only one, write copy** of the fragment.

Here, "read-only copy" means that the access right associated to the page is set to read. If a user wants to write into that same page, the CHORUS VM will handle the "artificial" write fault and will ask (using *mpGetAccess*) the mapper for the "write access right" on the page.

In order to achieve this, one needs to know which **fragments** of an object are on which sites and with which access rights *(Read/Write)*. So, with this information we are able to revoke the appropriate rights and/or data. The write access and/or data will be given to a site that wants to write a fragment that was shared readable between several sites. Similarly the read access and data will be given to a site that wants to read a fragment that was previously writable on another site, or readable on several other sites.

### 6.3 Server Architecture

The decentralized Server is, in fact, composed of multiple servers: one and only one Global Mapper and one Local Mapper per site. We will illustrate the basic architecture of the decentralized mapper, by describing what happens when a regular file is opened and accessed from two sites. The first site opens the file and writes a page. Then the second site, will open the same file and read that same page.

To perform an open, the PM sends a request to the MiX File Manager which loads the corresponding vnode in memory and generates a capability (named *real capability*) to access that vnode. The File Manager then sends an *mpAttach* request to the "**Global Mapper**" (**GlMp**) with the capability naming the vnode. The Global Mapper records that a new segment is now becoming active and associates a so-called **coherent capability** with the real capability. This capability is sent back to the OM, and then to the PM. The coherent capability is built in such a way that requests applied to this capability will be received by the Local Mapper of the site where it is used (see "Building a Coherent Capability" below).

When the PM has to perform a write operation, it applies the appropriate CHORUS system call to the capability it has received at open time. The CHORUS Virtual Memory generates an *mpPullIn* request that will be received by the Local Mapper. The Local Mapper will then direct this request to the Global Mapper, which in turn will forward it to the MiX File manager. From that point, the Local Mapper of the requesting site is the owner of the page, and known as such by the Global Mapper (see figure 4).

Thus, when a second site wants to read the same fragment of the same file, its Local Mapper sends a request to the Global Mapper, which replies that the (probable) owner of the page is the Local Mapper of the first site. Thus, the Local Mapper of the second site is able to request the access and the (up-to-date) data directly from site 1. Before replying, the Local Mapper running on site 1 needs to reduce the access rights of the local CHORUS Nucleus. This is done using the

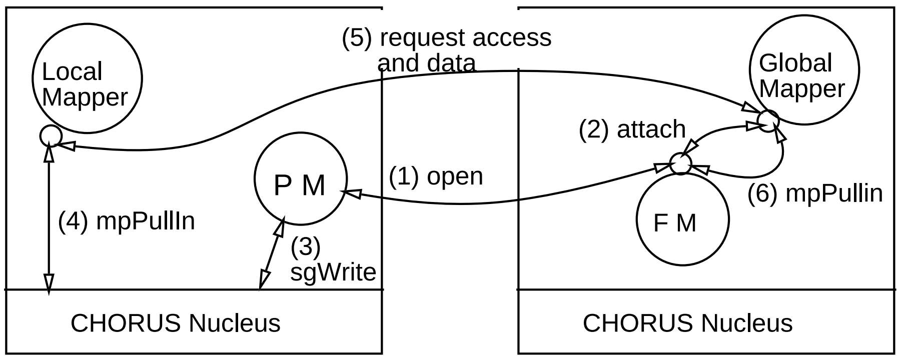

**Figure 4.** – First access to a filés fragment

*lcFlush* service, and generates, in turn, a *mpPushOut* of the modified page to the Local Mapper (see figure 5).

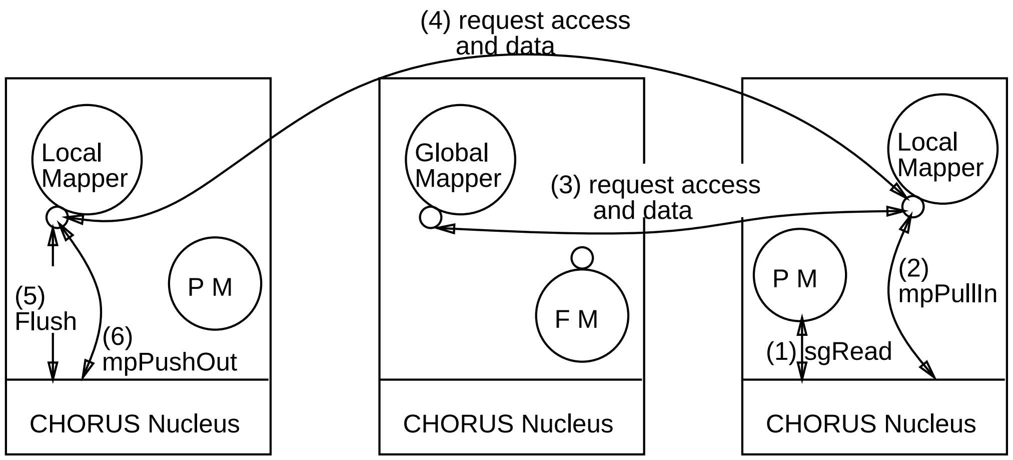

**Figure 5.** – Concurrent access to a filés fragment

A LoMp running on a given site will receive all of the requests sent by the CHORUS virtual memory of that same site. If the local mapper is the owner of the fragment it will be able to satisfy the request; otherwise it will have to send the request to the last known owner (or probable owner) of the fragment. The GlMp is the default owner of all the fragments of a segment. Thus, if a LoMp doesn't know anything about a fragment, it will ask the GlMp which will either reply with the data or will indicate which LoMp is the probable owner of a fragment.

All fragments have, at any given moment, a single local mapper which "owns" the fragment. A LoMp remains the owner of a fragment until a write access request is received, in which case, ownership is immediately transferred to the new writing site. The local descriptor for the fragment is updated to reflect the new owner. The next access requests received by an old owner will be re-directed to the new owning site. The LoMp has a knowledge of the fragments' probable owner, and the protocol used insures that the request will reach the owner of the fragment after a finite number of tries.

In this way, only the local mappers which share a fragment cooperate in order to maintain its consistency, thus increasing the degree of parallelism. Local mappers use an internal protocol to minimize the number of steps needed to find the owner of a fragment. Our algorithm reduces the number of local mappers to intervene and minimizes the data transfer from one site to another.

### 6.3.1 Probable Ownership

The performance of the distributed algorithm depends on how efficiently the information on a fragment probable owner is maintained. Fewer steps to reach the fragment owner will be necessary if the probable ownership information is "recent". In order to improve the accuracy of the knowledge of the "probable" owner by the local mappers, the probable owner field is updated with every communication between local mappers.

Figure 6 illustrates a possible access path from a LoMp, which performs a write access request (site 1), to the local mapper which owns the fragment (site 3). In the initial state the fragment information about the probable owner is the following: the Local Mapper of site 1 believes the probable owner is site 2, the Local Mapper of site 2 has its probable owner set to site 3 which is effectively the current owner of the fragment.

In figure 6.a LoMp 1 sends a write access request message to LoMp 2 (its probable owner). LoMp 1 will become the owner of the fragment because it asks for write access. Therefore LoMp 2 updates its probable owner field to LoMp1 before replying with LoMp 3 as probable owner. LoMp 1 now knows a new probable owner of the fragment (LoMp 3), and restarts its search until it finds the owner of the fragment (see figure 6.b).

In our example LoMp 3 is the owner of the fragment and the request implies owner change. So, LoMp 3 updates LoMp 1 as probable owner and replies to the the required access. When LoMp 1 receives the reply message from LoMp 3 it, stops the owner searching algorithm and becomes the owner of the fragment (see figure 6.c). The search for the owner is known to be finite (see[Li89a] ). For example, after step 8.b, every site requesting a fragment from site 2 will see its request re-directed to site 1 (not site 3) where it will be blocked until site 1 acquires the fragment.

## 6.4 Some Examples

Rather than describing the entire mechanism we will briefly describe some examples, illustrating how the distributed servers work.

### 6.4.1 Building a Coherent Capability

The first problem to solve is the naming issue. How to build a capability that will represent the coherent segment? We had one constraint to achieve this goal: the Local Mappers (LoMp) must be able to serve a request for an object they are not yet aware of. This avoids to modify the existing protocols between CHORUS/MiX V.4 servers. Moreover, in a Single System Image environment, processes may freely migrate from one site to another, thus the capability used to access a file in a coherent fashion must be such that there is no need to deal with Local or Global Mappers at migration time, so that migration can be kept as light as possible. This is achieved by constructing capabilities as described below (figure 7):

— The port on which the Global Mapper receives requests, is inserted in a static port group which has a unique 32 bit stamp. The port of the GlMp is the only one to belong to that group. This allows to name uniquely the mapper with a 32 bit long identifier.

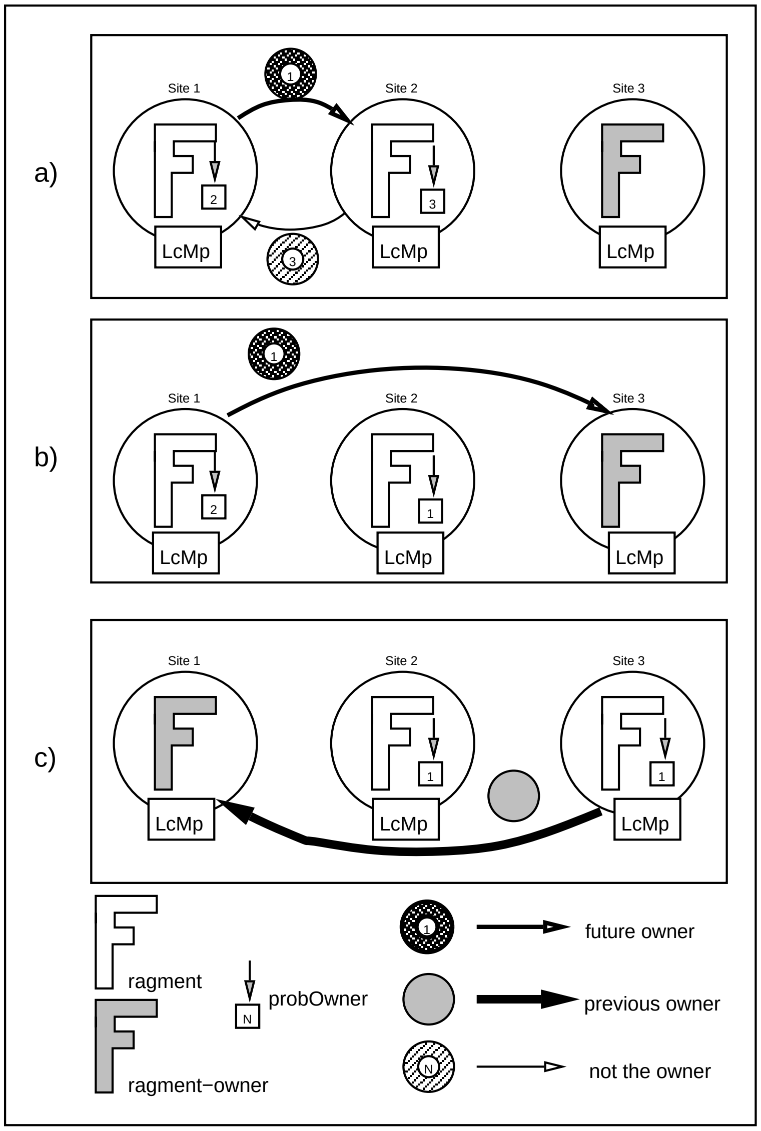

**Figure 6.** – Searching owner

— Thus a segment managed by the GlMp will be uniquely identified by its so-called "Global Capability (Gc)" built from the Global Mapper group stamp, and a Unique Identifier of the segment within the Global Mapper (e.g. address of the structure representing the segment).

— Each Local Mapper has a port on which it receives requests either from the local CHORUS Virtual Memory or from other Local Mappers. A group of ports, called the Local Mapper group, contains all these ports. Thus, on any site it is possible to reach the Local Mapper by using the name of that Local Mapper group with a functional addressing mode.

— The capability (Lc) used by a process to access (map, read... ) a coherent segment is built by the Global Mapper from: the name of the Local Mapper group, the Unique Identifier of the segment within the Global Mapper and the Global Mapper group stamp.

Any Local Mapper, using this capability is able to rebuild the "Global Capability (Gc)" to send a request to the appropriate Global Mapper for a segment.

| Rc | Lc | Gc |
|---|---|---|
| OM Porte | Grp. LoMp F | Grp. GlMp |
| Vnode | UID within the GlMp | UID within the GlMp |
| Key r | GlMp Stamp. | undef. |

**Figure 7.** – Segment capabilities

### 6.4.2 mpPullIn

In the *mpPullIn* example (figure 8), the initial state is the following: the fragment data is present only on site 1 and has been modified. In our algorithms the fragment data may be modified only on the site where the local mapper is the fragment owner, the LoMp 1 (i.e. LoMp 3 needs to acquire the write access first).

The virtual memory of site 3 wants to get the fragment data and the write access (*mpPullIn(W)*). The last probable owner known by LoMp 3 is LoMp 2, thus it sends a *dmpPullIn(W, site 3)* request to LoMp 2. *dmpPullIn* is the internal version of the *mpPullIn* request which is exchanged between Local Mappers. This protocol carries extra information allowing LoMp's to update their knowledge of the owner of a fragment.

LoMp 2 is no longer the owner of the fragment so it is not able to give the data and access of the fragment but it replies with its probable owner: LoMp 1. The LoMp of the site 2 updates its fragment owner notion: LoMp 3 because the access required was write. At this moment, the LoMp 3 must retry the operation with the next probable owner obtained. In this example the LoMp 1 owns the fragment, then it will be able to return the data and the required access.

The LoMp 1 invalidates the fragment in the virtual memory. This operation causes a *mpPushOut* request from the virtual memory to the local LoMp. The LoMp 1 records the new owner of the fragment: LoMp 3 before replying (*returnDmpPullIn*) with the data and the write access.

When the virtual memory needs physical memory space, it performs a *mpPushOut* request in order to write back the fragment modifications before freeing the corresponding physical page. So, it may occur that the data is no longer present (or cached) on the owner's site. In this case the owner LoMp will send a *dmpPullIn* request to the global mapper which will retrieve the data from the real server (e.g. MiX File Manager).

In the *mpPullIn* reply protocol a local mapper can indicate to the virtual memory that the given fragment has been modified but that these modifications haven't been sent to the global mapper. In this case the virtual memory will install the data with the dirty bit set.

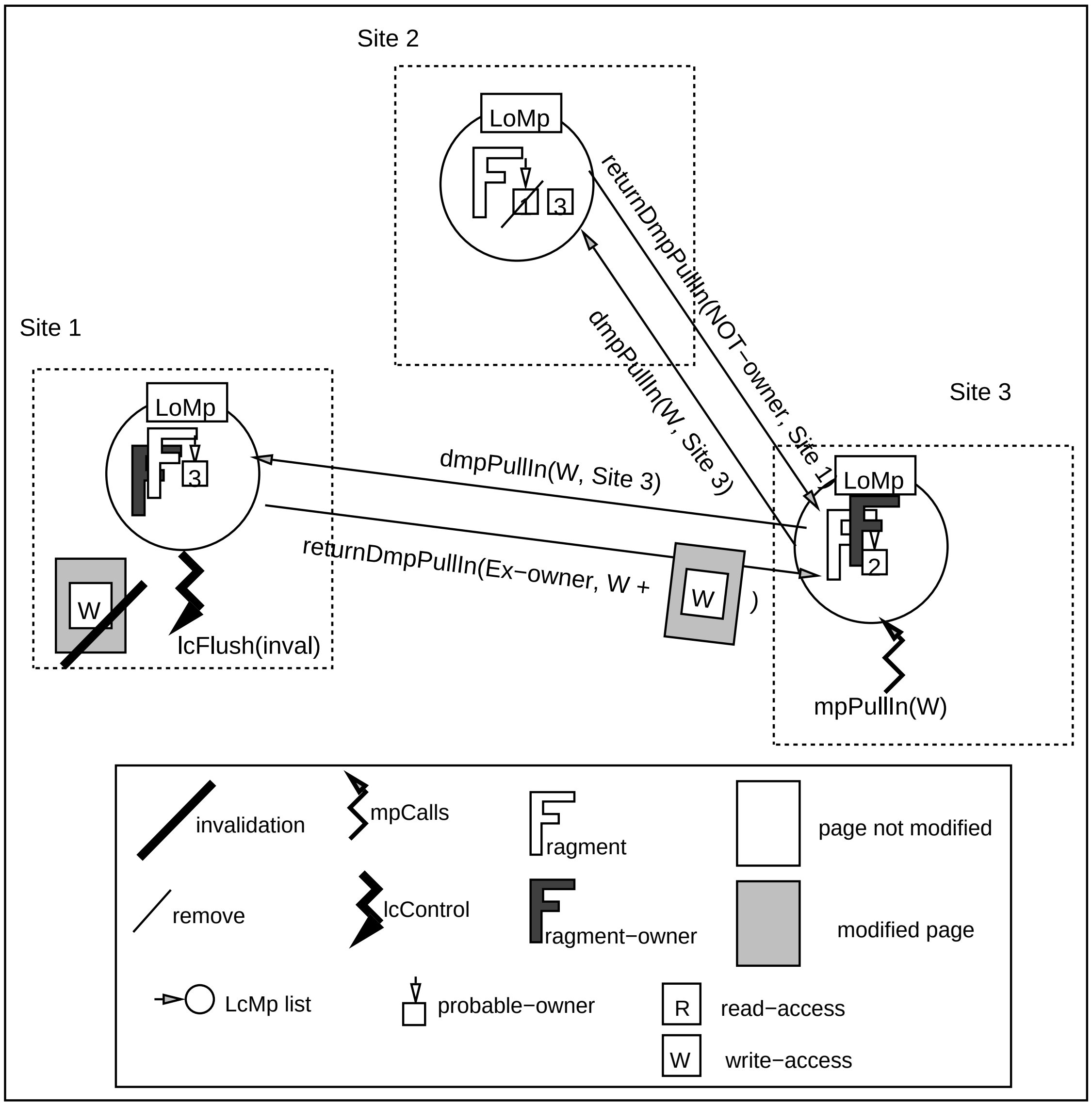

**Figure 8.** – PullIn example

### 6.4.3 Atomicity on large fragment

The local mappers must be able to process requests asking for access rights to a range of several pages. To solve this problem without deadlocks, we insure that pages will be acquired on any site in the same order.

For example, if two sites (I and II) ask for a commo fragment set, say, from page 1 to 3 of a segment, they will try to acquire page 1 first, without holding any other page of the desired set they may already have on their site. Therefore, if site I has get the ownership of page 1, before site II, site II will accept to give up the ownership of any page of the set until it becomes owner of the first page. When site 1 will release ownership on page 1, site II will be able to gain access to page 1, then to page 2 and finally to page 3. It cannot acquire access to page 2 before having gained access to page 1. Similarly, it cannot gain access to page 3 before having gained access to page 2.

## 6.5 Implementation Details

### 6.5.1 The GlMp

The GlMp performs segment management: it must maintain only one entry per real capability. The global organization tables are represented in the figure 9.a.

— *Hash segment table*: this is an array of segment list heads accessed via a hash function. Each list is protected by a mutex that is acquired only to walk through the segment list to search, add or delete a given segment. There is only one entry for a given segment.

— *Segment structure*: The segment structure fields are the following:

- *mutex B*: the second level of synchronisation. It is acquired for any operation performed against the segment descriptor.
- *RealCap*: this field contains the real capability of the segment.
- *Ref. counter*: the reference counter maintains the number of "attach" requests performed with the same real capability.
- *Granularity*: the granularity is fixed by the segment (i.e. it is the smallest possible size of a fragment).
- *head fragment list*: this points to the ordered, disjoint fragment list of the segment that have been requested by LoMp.
- *head LoMp's list*: it points to a local mapper list. Which contains all local mappers that have requested access to at least one fragment held by this global mapper.

— *fragment structure*: the fragment structure contains the description of a fragment. The components of this structure are the offset, number fragment (offset/granularity) and the access right granted to the probable owner of the fragment. The probable owner is the last LoMp that has requested a write access for that fragment to the GlMp.

### 6.5.2 The LoMp

The local mapper structures (see figure 9.b) are similar to the GlMp structures The overall structure is the same but a local mapper must manage other information in order to maintain fragment coherence.

- *hash segment table*: the structure is identical to the GlMp hash segment table.
- *segment structure*: this structure has some different information. Its fields are the following:
  - *GlobalCap*: the Gc (global capability) is rebuilt by the local mapper upon the first request received from the local CHORUS VM, it gives access to the GlMp segment description of the corresponding real segment.
  - *Granularity*: this information is obtained from the Global Mapper during the first *getAccess* or *pullIn*.
  - *MyCache structure*: this structure contains the local "local cache" associated to the coherent segment. There exist only one local cache per segment per site. Due to the preemptive scheduling provided by the CHORUS Nucleus, message passing in CHORUS can not guarantee the soundness of message ordering even though network protocols may insure it (i.e. messages sent from a port **p** to a port **p'** may arrive in a different order than they have been sent). This may cause an inconsistent situation; a *lcFlush* sent after a *mpPullIn* reply may arrive before to the VM. The *stamp* is used to solve a synchronization issue between the mapper and the CHORUS VM, as the VM guarantees to serve messages according to their stamp order.

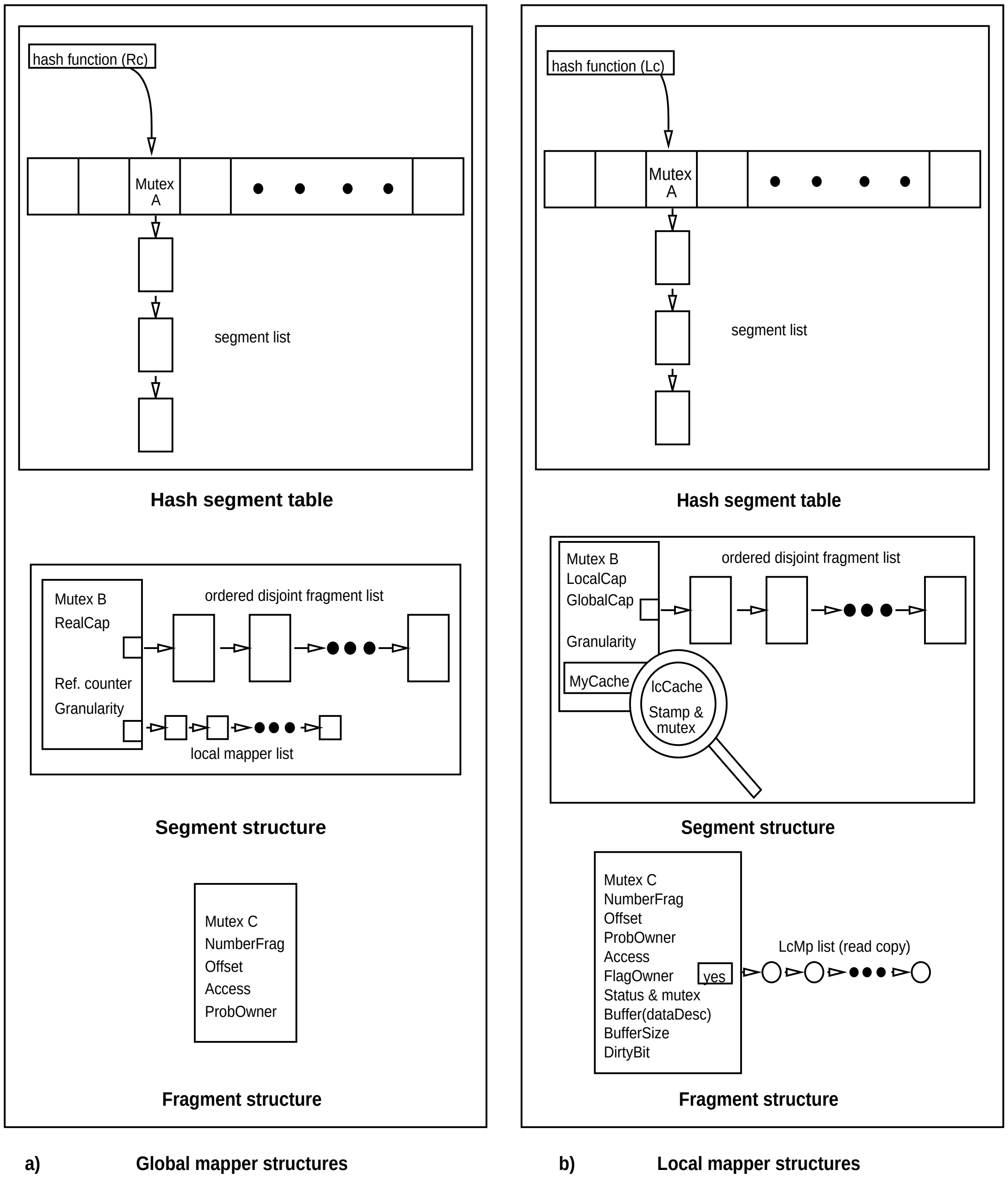

**Figure 9.** – Distributed mapper structures

- *Fragment structure*: this structure contains all information about the fragment.
  - *Mutex C*: this mutex permits requests on different fragment of the same segment to be run in parallel.
  - *NumberFrag*: as we use a fixed granularity per segment we can define a single number per fragment (offset/granularity).

- *Access*: is the fragment access right granted by the LoMp to the VM for this fragment. It may be WRITE, READ or NULL.
- *ProbOwner*: this field contains the information of the last owner as known by this LoMp.
- *FlagOwner*: the FlagOwner's value is "true" when this LoMp is the owner of the fragment else it's "false".
- *Head LoMp's list*: if the LoMp owns one fragment and there are several "read copies" of it in the network, this list contains the LoMps which have performed a *mpPullIn(R)* or *mpGetAccess(R)* operation on that fragment. This information is used at invalidation time.
- *Status & Mutex*: the Status value describes the pending request. This information is used when a *mpPushOut* request is received to determine whether the data should be written back to its storage (e.g.: *mpPushOut* corresponding to a UNIX sync operation) or sent to another LoMp (e.g. : *mpPushOut* corresponding to a *mpPullIn* operation). This allows to avoid writing back data to its storage when not needed.
- *Buffer(dataDesc)*: Reference to the data being moved that avoids mapping it into the mapper address space, and thus to manage the memory address space at each *mpPushOut*/*mpPullIn* operation.
- *DirtyBit*: this information is recorded at the same time as the body reference (*mpPushOut* time). If the data is "dirty" DirtyBit value is 1 else null. In the *pullIn* reply protocol we can indicate to a requesting LoMp if the data has not been written back after the last modification. The LoMp of the requesting site will install the page as already dirty.

## 7. Lessons, Status and Future Work

We have adapted the general description of the "dynamic decentralized" algorithms used in IVY to the CHORUS/MiX environment. This basis has been extended to:

— Support for UNIX semantics:
  - — It deals with secondary storage issue, allowing data to be written back to the storage server (i.e.: to the disk) or to be moved "dirty" from one site to another.
  - — It proposes a scheme to determine the initial location and owner of a fragment, without using a fixed partition mechanism. Such a mechanism would be meaningless in a Single UNIX System Image, where any node can access (or not) any file.
  - — Finally, it extends the basic mechanisms to solve the atomicity issue for concurrent read/write operations on overlapping fragments of a file.

— Independent reusable Server:
  - — The distributed consistency server is independent from both the CHORUS Virtual Memory and from MiX File Managers. Thus, it could effectively be used in any other environment provided that the protocols are implemented. Due to the notion of granularity associated with a segment, one could potencially use these servers to manage one byte long fragments. In fact, a 2 or 4 byte long fragment could be an integer managed in a distributed consistent way. In other words, our Distributed Consistency server acts mainly as a distributed "token" manager rather than as a Distributed Shared Memory Server.
  - — The coherency policy being implemented outside the CHORUS Nucleus itself, this allows some segments to be managed by a "strictly coherent" policy, while other segments may be managed by other mappers according to different schemes.
  - — The independence of the server from the nucleus and the subsystem has allowed us to develop it in parallel with the MiX V.4 subsystem, and to test and enhance the CHORUS Virtual Memory interface without being required to implement a full distributed UNIX V.4 system. Once again, micro-kernel and modularity has proven to be an efficient way

to develop system software.

The mapper has been implemented and a first version of it is running on COMPAQ 386 machines connected through Ethernet. It is coded in C++. The GlMp is 2500 lines of code and the LoMp is 4500 lines of code. Experiments are also being conducted on a distributed memory multicomputer platform. However, it is still too soon to show any performance figures.

In this first version, we have not paid much attention on the internal mechanisms that manage the ordered disjoint list of fragments. There is still work to be done to manage these lists in an efficient way that does not consume too much memory.
We expect the behavior of the server to fit most of UNIX applications requirements, but in some cases we believe that the scheme we use will not avoid thrashing situations, some futher work will be done to detect and solve these situations.

## 8. Acknowledgements

We would like to thank Michel Gien and Allan Bricker for their help in reviewing this paper and Bruno Pillard for his editorial assistance. Marc Rozier et Frédéric Herrmann participated in many of the discussions on the design and implementation of distributed shared memory in CHORUS.

## 9. References

[Abro89a] V. Abrossimov, M. Rozier, and M. Gien, ‘‘Virtual Memory Management in Chorus,’’ in *Pro. of the Progress in Distributed Operating Systems Management*, Springer Verlag, Berlin, (18-19 April 1989).

[Acce86a] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, ‘‘Mach: A New Kernel Foundation for UNIX Development,’’ *Summer Conference Proceedings 1986*, USENIX Association, (1986).

[Benn90a] Rice University, John Bennett, John Carter, and Willy Zwaenepoel, ‘‘Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence,’’ Technical Report, Texas, (Febuary 1990).

[Cher88a] David R. Cheriton, ‘‘The V Distributed System,’’ *Communications of the ACM*, vol. 31, no. 3, (March 1988), pp. 314-333. Vsyst

[Cohn91a] David L. Cohn, Paul M. Greenawalt, Michael R. Casey, and Matthew P. Stevenson, ‘‘Using Kernel-Level Support for Distributed Shared Data,’’ in *Proc. of SEMDS II Symp. on Experiences with Distributed and Multiprocessor Systems*, The USENIX Association, Atlanta, GA, (March 21-22, 1991), pp. 247-259. CS/EX-91-216

[Flei89a] Fleisch, Brett and Popek, Gerald , ‘‘Mirage: A Coherent Distributed Shared Memory Design,’’ in *Proc. of 12th ACM Symposium on Operating Systems Principles*, ACM SIGOPS, Litchfield Park, AZ, (December 3-6, 1989), pp. 211-223.

[Gray89a] Cary G. Gray and David R. Cheriton, ‘‘Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency,’’ in *Proc. of 12th ACM Symposium on Operating Systems Principles*, ACM SIGOPS, Litchfield Park, AZ, (December 3-6, 1989), pp. 202-210.

[Hell90a] SIEMENS, Hermann Hellwagner, ‘‘Survey of Virtually Shared Memory Schemes,’’ PUMA Working Paper n° 15, München, Germany, (August 1990), p. 64. CS/EX-90-346

[Li89a] Kai Li and Paul Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Transactions on Computer Systems*, vol. 7, no. 4, (November 1989), pp. 321-359.

[Mull87a] S. J. Mullender, *The Amoeba Distributed Operating System: Selected papers 1984 - 1987*, CWI tract 41, Amsterdam, (1987). AMO87

[Nitz91a] Bill Nitzberg and Virginia Lo, ‘‘Distributed Shared Memory: A Survey of Issues and Algorithms,’’ *Computer*, vol. 24, no. 8, (August 1991), pp. 52-60. CS/EX-91-287

[Orte91a] Maria Inés Ortega, ‘‘Distributed Shared Memory and UNIX: Experimentation in the CHORUS/MiX system,’’ CHORUS Systèmes, (September 1991).

[Pope85a] Gerald J. Popek and Bruce J. Walker, *The LOCUS Distributed System Architecture,* The MIT Press, Cambridge, MA, (1985), p. 148.

[Rozi88a] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Pierre Léonard, Sylvain Langlois, and Will Neuhauser, ‘‘CHORUS Distributed Operating Systems,’’ *Computing System*, vol. 1, no. 4, (October 1988). CS/TR-90-25

[Stum90a] Michael Stumm and Songnian Zhou, ‘‘Algoritnms Implementing Distributed Shared Memory,’’ *Computer*, vol. 23, no. 5, (May 1990), pp. 54-64.

[Tam90a] Ming-Chit Tam, Jonathan M. Smith, and David J. Farber, ‘‘A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems,’’ *Operating Systems Review*, vol. 24, no. 3, (July 1990), pp. 40-67. CS/EX-90-398

[Zhou90a] University of Toronto, S. Zhou, M. Stumm, K. Li, and D. Wortman, ‘‘Heterogeneous Distributed Shared Memory,’’ Technical Report CSRI-244, Toronto, Canada, (September 1990), p. 31. CS/EX-91-275

CONTENTS

LIST OF FIGURES