The Arias Distributed Shared Memory: an Overview

P. Dechamboux[1], D. Hagimont[2], J. Mossière[3] and X. Rousset de Pina[3]

[1] GIE Dyade (Bull-INRIA)
[2] Institut National de Recherche en Informatique et Automatique (INRIA)
[3] Institut National Polytechnique de Grenoble (INPG)

Unité de Recherche INRIA Rhône-Alpes 655, avenue de l'Europe
38330 Montbonnot St Martin - France

Abstract. Arias is a system serviceee for distributed information systems, which provides low-level support for applications that make an intensive use of sharing and persistence. Arias is based on a persistent distributed shared memory that can be tailored and tuned according to the needs of the supported environment. This paper summarizes the basic design choices and describes the implementation of the first prototype.

1   Introduction

Most current information computing environments operate on local or wide area networks. Such distributed environments are mainly used because they enable and enhance the cooperation of multiple users through specific distributed applications. Typical examples of such applications are large hypertext or hypermedia information management systems in areas like CAD and software engineering, or object-oriented databases. More precisely, this class of applications has the following characteristics:

- Large size of data, due to the world-wide spread of information bases.
- Persistence of processed data.
- Highly interconnected information, through multiple links.
- A high sharing rate among cooperating users.
- Data access from distributed workstations.

Information management systems must be based on current standards. One of such standards is Unix, despite the fact that this system is not adapted to provide support for these applications for the following reasons:

- Primitive mechanisms for information sharing.
- No integrated support for distribution, and particularly for naming.
- Persistence management by users.

The advent of 64-bit address processors and high speed networks allows us to reconsider the solutions currently proposed for applications implementation.

With fast networks, solutions based on distributed virtual memory become feasible and wide addressing spaces facilitate the management of a unique virtual space.

The goal of the Arias project is to design and to implement a set of system services which are characterized by the following points:

– Services should provide an appropriate support for sharing of complex and persistent data. They should meet the requirements of both cooperative applications and database managers.
– Compatibility with current standards must be preserved, specially with the Unix system.
– Services should benefit from recent technical evolutions in order to be efficient.

A large spectrum of existing runtime frameworks can benefit from these services (examples are given in section 7). Our main goal is not to implement applications directly above these services. We are rather aiming at providing services designed to support frameworks, each framework being dedicated to a class of applications with similar behavior. Already known examples are frameworks for distributed applications development and DBMS, which hide the low-level features of the system interface. In the following, we use the term *application* to describe client entities (frameworks or final applications).

After a general presentation of the provided services (section 2), we present respectively our proposition for memory management (section 3), consistency and synchronization (section 4), permanence (section 5) and protection (section 6). Section 7 presents our current experiments in using these services, section 8 contains a comparison with similar projects and section 9 concludes this paper.

2   General presentation

We want to provide a service which allows sharing of persistent data between Unix processes executing on homogeneous interconnected workstations. Three types of problems have to be addressed :

– Data sharing and consistency of shared data.
  If sharing is implemented by copying shared data on different machines, then we need to address the issue of the type of consistency to ensure between these copies.
– Resistance to node failures.
  Shared data is persistent, which means that its lifetime does not depend on that of the process which creates it. However, this property is not sufficient to resist node failures. For that purpose, we must provide a stable storage for permanent segment (for example on a disk). As a consequence, the system must also enable coherent (atomic) updates of the permanent image of segments.

– Access control.

We should allow access to shared data with different access rights according to processes or users; we therefore need to provide a protection service for shared data.

One of the basic design choices is to provide mechanisms which allow users to implement the policy which best fits the needs of their applications, rather than enforcing a system-defined policy. For example, the system does not enforce a consistency model, but provides the tools and the mechanisms which allow the implementation of different consistency policies. Application designers may also decide to use predefined policies which are provided.

Users may adapt the behaviour of the Arias system to the specific requirements of their applications by writing what we call a *specialized protocol*, and plugging it into Arias. In this way, they can parameterize the behaviour of consistency and synchronization (i.e., the management of the coherence of distributed execution memory) as well as of permanence (i.e., the management of the consistency of permanent memory with respect to execution memory).

The remaining of this section describes the service which handles data sharing, the service which implements permanence and the protection service.

2.1 Data sharing service

The service which handles data sharing is composed of two parts: shared memory management and consistency management.

Shared memory management

The main design choice for memory management is the use of 64 bit addressing for managing a single address space. In our system, shared memory is built out of a block of 2^{63} bytes reserved at the same virtual address in all Unix processes which use it. Therefore, virtual addresses may be freely exchanged between processes as system-wide unique identifiers.

The allocation unit within this memory is a segment, a sequence of contiguous pages. The segment size is fixed at creation time. Segments are persistent (i.e., their lifetime does not depend on that of the process which created them). A segment is designated by its base virtual address which remains unchanged during its whole life. Segments need to be explicitly destroyed. The system ensures that the range of address space occupied by a destroyed segment is never reallocated and that each attempt to access a destroyed segment generates an exception. The implementation of a garbage collector is left to the care of applications which are able to interpret the contents of segments.

During execution, shared memory management relies on the paging mechanisms of the machines. Due to segment sharing, concurrent accesses at the same page may occur simultaneously on several hosts, as pages can be replicated in the memory of each host. The system does not handle consistency among different copies of the same page, but it provides the mechanisms which allow applications to implement their own consistency policy.

Consistency management

The supplied mechanisms rely on the following observations:

- Most of the time, the segment and even the page have a granularity which is too coarse to be chosen as the synchronization and consistency unit. Hence, it is necessary to provide a finer grain unit if we want to avoid problems due to false sharing (processes accessing disjoint areas of the same page).
- Applications synchronize their access to potentially shareable data. It is therefore possible to link synchronization and data consistency enforcement, and specifically to postpone the application of consistency on a data zone to synchronization time (e.g. the request of a lock on this zone).

Mechanisms which implement consistency manage zones which are sequences of contiguous bytes. Each zone has a particular copy called the *master copy*. The system knows how to find the master copy of a zone at any moment; the system also provides primitives which allow updating zone copies from the master copy and conversely, or to change the ownership of the master copy. This set of primitives may be used to implement usual policies for consistency management (section 4). A number of "standard" protocols are provided, e.g. entry consistency protocol (strong consistency is only enforced at lock acquisition on zones) combined with a synchronization policy of readers/writer style and with a pessimistic transactional scheme.

2.2 Permanence service

A segment of memory may be made permanent (have an image on a permanent physical support, e.g. a disk), which permits failure resistance. Non permanent segments are called volatile. We provide two mechanisms for permanence management: permanent copy (make a segment permanent) and logging. Client applications can log a set of modifications of different segments as a list of records on disk. Validation of this log ensures that these modifications are atomic. In case of failure, the log allows a consistent image of the permanent memory to be rebuilt.

The format of modification records and the processing of these modifications are handled under applications' control. Actually, application control the format and the semantics of records written into the log, along with the use of the log upon error recovery. An application can therefore manage a "before" log (store old validated values) or an "after" log (store validated modifications). The application also controls the updates of permanent images of segments which can be modified implicitly in an asynchronous way, or explicitly in a synchronous way.

In our system, we distinguish between two levels of memory: the execution memory where segments are manipulated by applications, and the permanent memory. This distinction allows us to always maintain, even during execution, a consistent version of segments in permanent memory.

To facilitate their management, permanent images of segments are grouped into logical storage units called volumes. A volume is managed by a single server at a given moment. It can be transferred to another server during its existence.

2.3   Protection service

The protection service provides facilities for controlling accesses to segments. The protection service is based on two concepts: protection domains and capabilities. A protection domain defines a set of accessible segments along with the available operations for each of them. A system mechanism ensures that processes executing within a domain access its segments according to the specified rights. A capability is a data structure which defines an access right on a segment. It integrates a segment name (virtual address) and a set of access rights to this segment, expressed in terms of read, write and execute operations.

At a given moment, a Unix process using the protection service runs within a protection domain which defines its current addressing context. Two different protection domains can share segments with the same or different rights. At the first access to a segment within a domain, the system checks if the domain owns a capability for this segment, and if so, authorizes the mapping of the segment with the specified rights.

A domain can export capabilities of a special type to other domains. Such capabilities are called *domain capabilities* and define the entry points of the domain. A domain capability is associated with a procedure to be executed in another domain. When such a procedure cannot be executed locally because the segment to which it belongs cannot be mapped locally, the domain capability is searched for and a cross-domain procedure call is performed if it was found.

A domain change generally includes a transfer of capabilities from the caller to the callee for input parameters and conversely for the output ones. The protection service provides an interface description language (IDL) which allows the transferred capabilities to be specified. Capability transfers are specified by the domain which exports its entry points and must be accepted by the caller (cf. section 6). The advantage of this approach is that the executed code can be rendered totally independent of protection.

In the following, we essentially study the four basic concepts necessary for the implementation of the memory service which we have designed. These are segment management, consistency and synchronization, permanence, and protection.

3   Segment management

Persistent memory is composed of segments. A segment keeps its base virtual address unchanged during its whole lifetime and also keeps the same size, fixed at creation time.

3.1 Manipulation of segments

Creation. In order to create a segment, an application provides the size of the segment to allocate, and obtains in return its base address. Only virtual space is allocated at creation. Data pages which physically compose the segment are created only when accessed by applications.

Destruction. The destruction of a segment is always explicit. An application can make it implicit at its level by providing a garbage collector. Segments manipulated by the protection service are destroyed when the capabilities which designate them disappear from the system. Explicit destruction of a segment is immediate and irrevocable, but it may require special rights if the segment is protected.

Destruction of a segment implies the destruction of all of its associated structures, including its permanent image, if such an image exists. After the destruction of the segment, the range of virtual space it occupied becomes invalid. Some applications may keep pointers which point into this address interval, or certain segments may contain such pointers. Attempts to access a destroyed segment raise an exception in the calling application.

Access. First access to a page by an application triggers a page fault. To process this fault, the system first identifies the segment to which the missing page belongs. If the system discovers that this page does not belong to any segment (a segment which has not been allocated yet, or an already destroyed one), an exception is raised. Otherwise, the segment is mapped at the host (if not already mapped), and a copy of the page is loaded on the faulting node (see section 4).

Since specific protocols are managing data consistency (more precisely zone updates), the system pager is not responsible for loading consistent data in physical memory (it mainly allocate pages).

3.2 Location

We consider here the problem of locating segments in virtual memory. Recall that we distinguish two levels of memory: execution memory and permanent memory. A segment is in execution memory if it is used by at least one process. If a permanent segment is not in execution memory, the system must locate it in permanent memory, and then create the data structure which represents it in execution memory. A volatile segment exists only in execution memory.

A segment is represented in execution memory by a segment descriptor, which contains the necessary information for data consistency management. The initial assumption is that this descriptor is managed in a single version at a single host, called *primary host* of the segment. All the hosts which use the same segment should communicate with its primary host to access the descriptor. We show later that we can avoid systematic communication with the primary host through caching mechanisms.
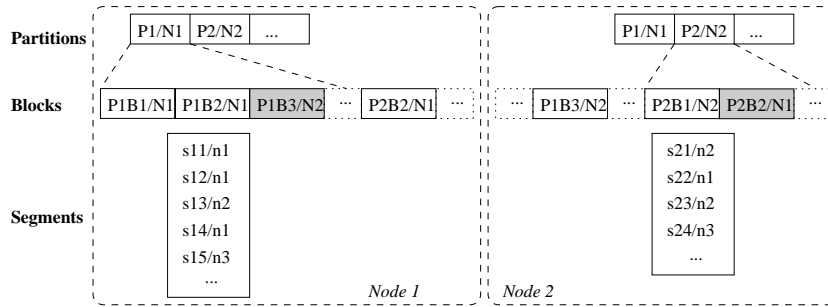
Fig. 1. Location scheme

In order to locate the descriptors (and thus the segments) that are not present locally, we implement the mechanisms depicted in Fig. 1. On each node, there is a table which allows to locate the primary host of a set of segments (the segment level in the figure). This association is defined on a unique node for each segment. Arias uses a two-level decoding scheme in order to locate this node:

– Partition Table.
  The global virtual space is sliced into partitions of equal size that can be dynamically allocated when more virtual space is needed. They have a "location host" which is responsible for locating directly or indirectly the table containing the primary host of the segments belonging to its virtual address range. The partition table is replicated on all sites and is maintained strictly consistent.
– Partition Blocks.
  In order to improve segment allocation (see section 3.3), partitions are sliced into blocks of equal size. A block of a partition can be managed either by the partition host or by any other host. The blocks table of a partition, located at the partition host, gives the host that manages a block. This host always contains the table entry which defines the primary host of each segment belonging to this block address range. Thus, finding this entry costs at most two messages: one to the partition host and one to the block manager host.
  Finally, each host which uses a segment but which is not the primary host of this segment owns a *descriptor cache*. This cache contains the location of the primary descriptor along with other information.

The segment descriptor can be moved for efficiency or administration reasons. In this case, the location host is notified, along with all the hosts which carry a descriptor of this segment in their cache. This systematic update is usable because the displacement of a descriptor is a rare event compared to the cost of these notifications.

3.3 Allocation of segments

When a segment is created, it is necessary to carefully choose its base address because this address is immutable, and because it identifies the segment and allows the location host to be located.

When the creation of a segment is requested, the application can optionally specify in which partition it wants the segment to be created. If such a constraint is not expressed, the system chooses a partition according to the load of the location hosts.

The simplest solution to perform allocation is to maintain an allocation cursor for each partition and to send allocation requests when partitions are remote. To avoid messages, we use a two-level allocation scheme (Fig. 1): at the partition level, we allocate *allocation blocks*; then, inside these blocks, we allocate segments. Allocation blocks are address intervals of intermediate size between that of partitions and that of segments. Each host owns an allocation block belonging to each of the existent partitions of the system. It allocates segments in the blocks belonging to the relevant partitions, without the need to contact the location hosts of the concerned partitions. When a block is exhausted, it is returned to its home location host and a new block is taken.

By using this technique, we reduce communications between the host which wants to create the segment and the location host of the partition in which the segment needs to be created. This allows a very fast creation of segments and also allows using Arias for data which has a very short lifetime.

4 Consistency and synchronization

Shared virtual memory systems generally impose on applications a unique service for consistency and synchronization, which may not be appropriate for their needs. As no universal model for consistency and synchronization (C&S) has emerged, applications must support the cost of an unadapted service. With Arias [11], applications have the possibility to define their own model, or to choose one among several available models.

In this section, we present this approach and its rationale, then we describe the architecture of our service of C&S and the functions it offers.

4.1 The Arias approach

We realize the C&S service in two layers: a "generic" layer which implements the functions of a C&S service which are independent of any particular model, and a "specific" layer which implements a particular C&S protocol. A specific C&S protocol is implemented using the functions of the generic layer. In the generic layer, we find the definition of the consistency and synchronization units and their location in the system. The C&S unit which we manage is the *zone*. A zone is a sequence of contiguous bytes, without limits on its size, but included in a segment (see section 3). A zone is designated by its initial address and specified by its address and its size.

Each zone has a master copy which resides on a particular host called the *zone master*. The master host makes all decisions concerning the aspects of the zone (number of copies, mobility, etc.). At a given moment, each zone has a master host; this master host can change dynamically. The location of a zone in the system is handled by the zone master.

When an application wants to access a zone, it asks the specific C&S protocol associated with its segment; if the protocol sees that the host which has issued the request is the zone master, it handles the request locally. Otherwise, the demand is forwarded to the master host, the location of this master being the responsibility of the generic layer.

The principal function of the generic C&S layer is to provide an association between a zone and its master in a transparent way for specific protocols. The interface of the generic layer is the same as that of a message passing layer. The generic layer allows the emission of a message to the master of a determined zone and the reception of the answer. Elements of the generic layer and its architecture are detailed in the next section.

## 4.2   General architecture of the C&S service

The generic C&S layer is implemented on each host. Above it, we implement the specialized protocols layer. The generic layer routes C&S messages to the concerned zone master and route answer messages. These messages contain two types of information.

- C&S information to be interpreted by the specialized protocol.
  It can be a synchronization request, but also the contents of a zone when managing data consistency.
- Information which concerns exclusively the generic layer.
  This information includes the zone concerned by the message, the specialized protocol involved and information about the message nature (displacement of the master copy, emission of a copy of the zone, answer to a message).

In order to reach the zone master, the generic layer asks the segment location service (section 3) for the primary host of the segment to which the zone belongs. On the primary host, the segment descriptor keeps the association between a zone and its zone master which allows the generic layer to know where it must route the message. On each host, there is a cache which contains the identity of zone masters that have been accessed recently. These caches are fragments of the descriptors of mapped segments. They are lazily updated on cache misses. With each specialized protocol is associated a message handler. This handler is activated upon the reception by the generic layer of a message concerning this protocol. The handler interprets the message and executes appropriate actions. The generic layer allows the registration of new protocols. It associates a unique identifier and keeps the association handler-identifier of the protocol.

5 Permanence

Some of the segments of the distributed virtual memory can be made fault resistant. This fault resistance is obtained through managing a permanent image of the segments on a stable support like a disk. In this section, we study interaction between the segment image in execution memory and the one in storage memory. Like other Arias services, the goal is to provide applications with a set of mechanisms allowing to build different storage policies. The interest of having several policies is that they can provide different levels of reliability and of efficiency.
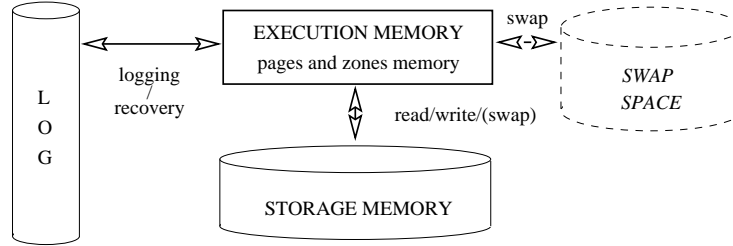
Fig. 2. Architecture of the storage sub-system

Fig. 2 displays the different elements which compose the storage service. This figure gives a purely centralized vision of the storage service; distribution will be considered later. Our system manages two disjoint and independent storage spaces: the storage memory and the log.

We now provide a detailed description for each of these elements and explain how the execution memory needs to interact with them (e.g., optimization of the paging process).

5.1 Storage memory

The storage memory contains the permanent images of segments. It is composed of the storage memories of the hosts participating in the implementation of the system. Each storage memory is divided into volumes in which permanent images of segments are stored. A permanent image of a segment is fully contained within a volume. Manipulations which can be done on the storage memory of a host are very limited:

- Creating and deleting volumes and permanent images of segments on disks.
- Reading and writing a page, or a part of a page in a segment.
- Making a permanent segment unavailable (it cannot be mapped into the virtual memory anymore).

The only properties guaranteed by the permanent memory are atomic creation or destruction of the permanent image of a segment, along with the atomicity of the modifications of a page which belongs to a segment. However, atomicity for the modifications of a segment relatively to a set of modifications is not guaranteed. If an application requires such a property, it should implement it through the logging mechanism described in section 5.2.

The storage memory service can be used by applications in order to explicitly copy modified parts of permanent segments. This service is also used by other components of our system. For instance, an asynchronous copier uses it to copy zones of permanent segments with validated modifications (i.e., modifications which have been previously registered into a log).

In principle, we manage two levels of independent memory: execution memory and storage memory. Theoretically, no cooperation is needed between the managers of these two memory levels. However, for efficiency reasons and to spare paging space, the paging manager relies on the storage memory manager to store permanent segments pages. This cooperation is described in section 5.3.

5.2   The log

The system provides a logging service which ensures the atomicity of a set of modifications that have been done on permanent segments. This service allows applications to register data into a log so they can repair permanent segments after a failure.

The logging service is expandable as it allows the use of several recovery protocols. These protocols are used upon a node failure to repair the part of permanent memory managed by that host. During the failure recovery, the logging service is able to determine whether a log has been validated or not. It then applies the appropriate recovery procedure from a particular protocol, the records of a log being related to one of these recovery protocols.

The actions associated with the logs are log creation, log records registration and log validation or abortion.

A log is a set of ordered records, all related to one application. These records are written in sequential order by an application; the order of writes defines the order of records. They are composed of a header whose structure is known by the logging service followed by information dedicated to a particular recovery protocol. The structure and the semantics of the second part of information are not known by the logging service. It is only processed upon failure by the relevant recovery protocol.

It is possible to consider two implementations of logs. Both have an influence on the efficiency of logging and the availability of permanent segments upon failure. We present these implementations below:

Centralized. In the first implementation, a log is stored on a single host. This provides a total order for operations to be executed at error recovery.

Distributed. The second implementation, which has been chosen, manages distributed logs. Such a log is partitioned, and therefore it becomes necessary

to define the ordering conditions. The partition is done in such a way that each log partition is stored at the node where the concerned segments reside. Therefore, such a node can recover almost autonomously. In this solution, a total order is ensured for a log partition. At the global level, only a partial order is ensured but it is sufficient.

The logging service must be very efficient. The handling of I/O operations (mainly updates under normal conditions) constitutes the major source of performance hits. It is obvious that sequential access to the disk reduces the I/O cost as it limits the movement of the disk head. The solution that we suggest is therefore to realize logical logs over a physical log. This physical log is stored on a dedicated disk if we want to keep a maximum efficiency, or on a special partition of a general-use disk. Each host which participates in the implementation of the permanent memory owns therefore a physical log.

Finally, an asynchronous copying service is implemented so that updates on permanent images of segments can be saved on the permanent image without provoking a performance hit. A mechanism which manages the consumption of the physical log is coupled to the asynchronous copier. This coupling allows the elimination of obsolete logs from the physical log so the storage space they occupy can be reused.

5.3   Paging strategy

An objective of Arias is to adapt the paging strategy applied at each node to the requirements of the system. This strategy does not take into account the distributed context of Arias. Decisions are always made locally and aim at minimizing expensive disk I/Os and swap space consumption.

Strategy for volatile segments. Pages belonging to volatile segments have always an associated image reserved in swap space. The only optimization that can be applied here is to avoid to copy the page at swap out time if both copies are already equal.

Strategy for permanent segments. An associated image is reserved in swap space for pages belonging to permanent segments only when the page is modified on the site. At swap out time, nothing is done for unmodified pages as they can be reloaded from storage memory. For other pages, if the page is valid with respect to modifications (i.e., all local modifications have been validated), the page is copied to the storage memory and swap space can be released. In other cases, the page is copied to swap space.

6   Protection

The protection mechanisms that we propose must allow different protection policies to be implemented. In particular, we do not want to be restricted to a hierarchical protection model. Given that, we decided to use protection mechanisms based on software capabilities.

Our protection service [7] is based on the classical concepts of capabilities and protection domains. A capability represents rights allowing either to access a segment or to change the protection domain. A protection domain is defined as a set of capabilities.

We also decided also to separate the definition of protection policy from the application development. This means that capabilities are transparent to the code which uses Arias services. Solutions where cooperating protection domains exchange capabilities explicitly to allow this cooperation were also excluded. Domain changes are transparent for the code which triggered them, so that the code does not need to manipulate other things than virtual addresses. Domain changes are performed implicitly when required. It allows an existing application to be protected without any modification on the application code.

To implement this transparency, capabilities are managed only by the system. When an application program addresses a non-mapped segment, the process traps into the kernel where a capability associated with that segment is searched for in the current domain. This capability can authorize the segment to be mapped within the domain, or it can allow the execution to be transferred to another protection domain. This way, the program manipulates only virtual addresses and its execution depends only on the content of the protection domains.

In the remaining of this section, we present the implementation of the protection domains, capability management, and the implementation of domain changes.

6.1 Implementation of domains

A domain is a protection space which can be shared by several Unix processes that have the same rights on the domain segments.

In order to make domain management more flexible, we found essential to allow the management of capability lists. A capability list can represent a set of rights to a service. These rights can be given by providing simple access to the list, which implicitly gives the rights defined in the list. Access to these lists is also controlled through a capability.

A capability can therefore give rights on a segment, on a domain (domain change) or on a capability list. An addressing fault on a segment triggers a look up for a capability asosciated with it in the current domain.

As explained before, each user level Unix process initially executes within a protection domain. However, such a process may change its protection domain. Thus, the execution flow of control (also called activity) needs to migrate to another addressing space, associated with the target domain, in which only authorized segment will be made available. This addressing space is implemented with a Unix process which behaves as a server for that domain. This domain server is dynamically created if it does not exist yet on the site where the domain is invoked.

Notice that protection and distribution are orthogonal since a domain change can always be performed locally.

6.2 Capability management

Capabilities are managed by the system. This means that it is not possible for a program to forge a capability. All capabilities are registered by the system and a new capability is created at segment creation time.

The system handles three types of capabilities:

- Capabilities on segments.
  They allow a segment to be mapped in the domain which contains the capability. The capability also specifies the mode of the authorized mapping (read-only, read-write, or execution).
  When an activity tries to access a segment which has not been mapped or which has been mapped with a non compatible mode, the system looks for a capability on the segment with the appropriate mode. If such a capability is found, the segment is mapped in this mode; otherwise, it is a protection violation and an exception is sent to the activity which has triggered the fault. A capability on a segment is composed of the segment address and of the authorized mapping mode.
- Capabilities on capability lists.
  Capability lists are designated by means of unique identifiers. They provide the rights specified by the capabilities they contain. These capabilities can be on their turn placed into lists, allowing the construction of capability trees. Trees and lists are used to specify the content of a protection domain. A capability on a list defines rights to read (gives the rights specified by the capabilities contained in the list), to write (allows addition or removal of capabilities) or to copy (allows the copying of a capability from the list).
- Domain capabilities.
  A domain capability is a capability associated with a procedure call. It is used when a code segment (a segment that contains procedures) cannot be mapped into the current protection domain. The capability allows the activity to be transferred into another domain where it can call the procedure. A domain capability is composed of a domain identifier, of an entry point address (procedure) and of the specification of parameters transfer rules.

6.3 Domain changes

When a program changes its domain on a procedure call, the procedure's actual parameters are passed to the target domain. The execution of the procedure in the domain might need capabilities associated with these parameters, in particular when parameters are pointers to shared segments. The problem is that of the transfer of these capabilities without any modification on the application code.

In order to solve this problem, we associate with a domain capability the specification of the corresponding procedure call. This specification describes the parameters and the control over associated capability transfers between domains.

To specify these domain entry points, we use an IDL (Interface Description Language) extended with clauses related to protection. This language allows the

definition of PPI (Protected Procedure Interfaces). A PPI defines the signature of a procedure, and for each parameter it defines the manipulation rules for capabilities to be applied when a call to this procedure triggers a domain change.

The protection rules should permit the control of both entering and leaving rights for the caller and the callee domains, and upon the call and the return of the procedure:

- The server specifies the capabilities required to perform the call and the capabilities which it can give on return of the call. Our extended IDL also allows to specify where incoming capabilities should be placed in the domain capability tree.
- The client specifies if it accepts to give the required capabilities and if capabilities returned are sufficient (Actually, the client accepts or refuses the PPI of the server); it also specifies the place for the returned capabilities to the caller domain.

When a server creates a domain capability, it associates with it a PPI which corresponds to its service. A client can either accept the capability as is, or copy the domain capability to overload the PPI definition (to change the place for received capabilities).

The extended IDL allows the specification of the capabilities that must be passed along with pointer parameters, the type and attributes of the capabilities (r/w/x) and the placement of the capabilities. Below is an example of PPI:

```
procedure Biblio_Register (
  in capa_seg_read TDOC *doc
     install BIB_DOC_LIST,
  out String[10] refbib
  );

procedure Biblio_LookUp (
  in String[10] keywords,
  out capa_seg_read TLISTREF *listref
  );
```

With the first PPI, a read capability is passed with the *doc* parameter. This capability is to be installed in the *BIB_DOC_LIST* capability list in the target domain. With the second one, a read capability is returned. If the client accepts the domain capability with this PPI, the returned capability will be added to a temporary list associated with the activity. However, the client may copy the domain capability and overload the PPI associated with the new capability:

```
procedure Biblio_LookUp (
  in String[10] keywords,
  out capa_seg_read TLISTREF *listref
     install BIB_LIST_REF
  );
```

This way, the returned capability will be installed in the capability list designated by *BIB_LIST_REF*. The IDL also provides the possibility to specify the transfer of a set of capabilities associated with a complex data structure. For example, the following PPI allows a set of capabilities associated with an array to be passed in parameter:

```
procedure Proc (
  in (capa_seg_read char *)tab[10]
  );
```

A similar PPI can be defined for a link list structure.

7   Experiments

As we have already emphasized, the integration with the Unix system is a key design choice of the project. We have implemented our platform over the AIX system, a version of Unix running on IBM RS-6000 and Bull Escala Machines. We are currently experimenting with our prototype with the following applications:

- A Clustered File System (CFS).
  We have implemented a distributed file system based on our distributed shared memory. This file system uses distributed shared segments in order to share files between the workstations of multiple users. We have implemented several consistency protocols in order to evaluate the advantage of tailoring specific protocols according to the application needs. Our results show that CFS out-performs the native NFS implementation on our platform.
- An object database system (O2).
  We are currently experimenting with object-oriented databases. More precisely, we are implementing a new version of the O2 [4] Store database system based on the Arias distributed shared memory. Such database systems are often implemented above Unix, which means that they must develop persistence and data sharing services on their own. Preliminary results show that we can reduce the overhead of these services in a sensitive way.
- CAD tools.
  A very promising experiment is the implementation of distributed cooperative CAD tools on top of a distributed shared memory system. We are currently involved in a European project in which CAD tools from the building industry should be ported on a platform similar to Arias built out of the technology provided by the project partners.

8   Comparison with existent models

In this section, we compare our works with projects of the same field. For this comparison, we consider the four work axis we described in sections 3, 4, 5 and 6.

Management of a single virtual space Several research projects are considering the use of 64-bit processors to manage a single virtual space in a distributed system. This is the case of Opal [3], Mungi [8], and Angel[10]. The use of virtual addresses as a unique name is widely recognized, but very few solutions have been proposed for address allocation, and for the location or the migration of segments inside this space. This is one aspect where we intend to provide new solutions. Besides, these projects provide solution which are not compatible with Unix as processes cannot directly address their private data.

Consistency and synchronization The strong coupling between consistency and synchronization is inspired from the work done on cache consistency maintenance in distributed DBMS [5] and in the Midway project [1], where Entry Consistency is used.

Management of different consistency protocols has already been addressed in research projects like Munin [2], but no current system allows applications to specify their own protocol for consistency management. Munin allows only a limited choice among a set of protocols implemented as a part of the system.

Permanence As for consistency, current systems [9] do not allow applications to specialize the management of logs in function of their particular needs. We provide applications with the possibility to ensure the permanence of their segments the implementation of an optimal logging policy.

Protection Most systems based on the use of a single virtual address space provide protection through software capabilities and protection domains (Opal [3], Mungi [8]). However, these systems impose on users the direct manipulation of capabilities and use encrypted capabilities. Therefore, application programmers are responsible for a part of capability management. We suggest that capabilities should be handled in a transparent way for users, which makes applications code independent of the protection.

9   Conclusion

In this paper, we have presented the objectives and the overall design of the Arias distributed shared memory system.

The Arias system manages persistent shared data in a unique address space, thus allowing pointers to be shared between distributed cooperating processes.

The system provides a service which allows these data to be managed on a stable storage and to be updated atomicaly, thus resisting node failures.

Arias also provides a protection service based on software capabilities for access control on shared data.

The main rationale behind those services is not to implement policies, but rather to provide generic services from which the most adequate policy can be delivered.

The evaluation of the current prototype is not yet completed, but preliminary results have validated most of our design choices. The perpectives of continuation of this work is first to pursue the experiments with the distributed

environements described above. Also, we found that most of the ideas applied in our framework could be applied to other existing systems. In particular, the protection model have already been implemented in a CORBA architecture and in the Java environement [6].

References

1. B. Bershad and M. Zekauskas, "The Midway Distributed Shared Memory System" COMPCON'93 Conference, pp. 528-537, February 1993.
2. J. Carter, J. Bennett and W. Zwaenepoel, "Implementation and performance of Munin", *13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pp. 152-164, October 1991.
3. J. S. Chase, H. M. Levy, M. J. Feeley and E. D. Lazowska, "Sharing and Protection in a Single-Address-Space Operating System", *ACM Transactions on Computer Systems*, vol. 12, num. 4, pp. 271-307, November 1994.
4. O. Deux et al., "The O2 System", Communication of the ACM, 34(10), October 1991.
5. M. Franklin and M. Carey, "Client-Server Caching Revisited", Proceedings of the International Workshop on Distributed Object Management, Edmonton, Canada, August 1992.
6. D. Hagimont, J. Mossière and X. Rousset de Pina, "Hidden Capabilities: Towards a Flexible Protection Utility", Proceedings of the 7th SIGOPS European Workshop, September 1996.
7. D. Hagimont, J. Mossière, X. Rousset de Pina and F. Saunier, "Hidden Software Capabilities", proceedings of the 16th International Conference on Distributed Computing Systems, May 1996.
8. G. Heiser, K. Elphinstone, S. Russell and J. Vochteloo, "Mungi: a distributed single address-space operating system", Proceedings of the 17th Australasian Computer Science Conference, pp 271-280, January 1994.
9. C. Mohan, D. Haederle, B. Lindsay, H. Pirahesh and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", ACM Transaction on Database Systems, 17(1), March 1992.
10. K. Murray, A. Saulsbury, T. Stiemerling, T. Wilkinson, P. Kelly and P. Osmon, "Design and implementation of an object-orientated 64-bit single address space microkernel", Proceedings of the 2nd USENIX Symposium on Microkernels and other Kernel Architectures, pp. 31-43, September 1993.
11. E. Pérez-Cortés, P. Dechamboux, T. Han, "Generic Support for Consistency in Arias", Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HOTOS-V), pages 221-226, Rosario, Washington, May 1995.

Table of Contents

This article was processed using the LaTeX macro package with LLNCS style