Distribution and persistence in multiple and heterogeneous address spaces

Paulo Ferreira, Marc Shapiro

▶ **To cite this version:**

Paulo Ferreira, Marc Shapiro. Distribution and persistence in multiple and heterogeneous address spaces. [Research Report] RR-2016, INRIA. 1993. inria-00074655

HAL Id: inria-00074655
https://inria.hal.science/inria-00074655v1

Submitted on 24 May 2006

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

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

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

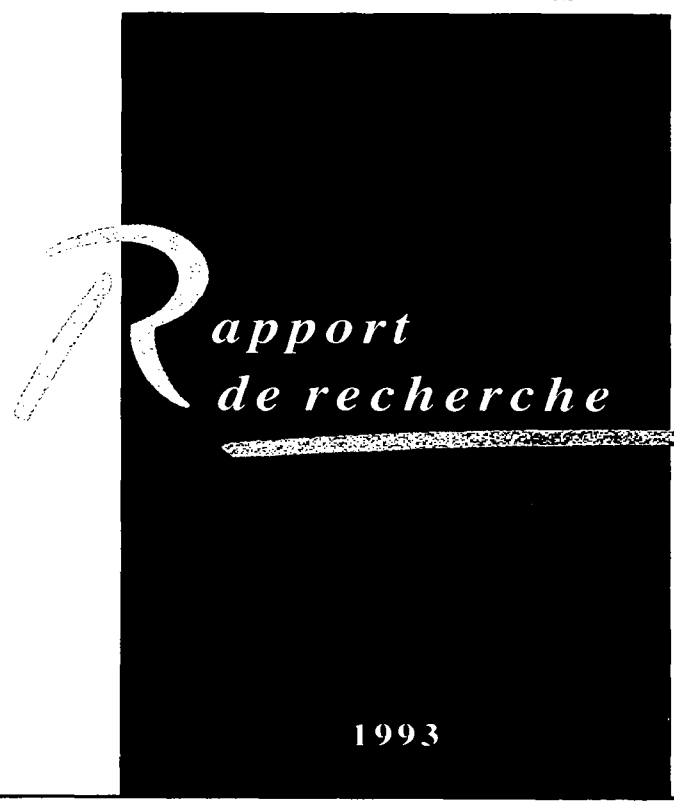
Distribution and Persistence in Multiple and Heterogeneous Address Spaces

Paulo FERREIRA
Marc SHAPIRO

N° 2016
Septembre 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués



Distribution and Persistence in Multiple and Heterogeneous Address Spaces*

Distribution et Persistance dans les Espaces d'Adressage Multiples et Hétérogènes

Paulo Ferreira †
Marc Shapiro

August 1993

*This research was supported in part by Esprit Basic Research Action BROADCAST 6360.
†Supported by a JNICT Fellowship of the Program Ciência (Portugal). Also affiliated with the Laboratoire d'Informatique Théorique et de Programmation, Université Pierre et Marie Currie, Paris.

Abstract

We present the design of a flexible architectural model that supports clustering, storing, naming, and accessing objects in a large scale distributed system. The system is logically divided in zones, i.e., groups of machines with an homogeneous address space organization. Both uniform (64-bit) zone-wide and partitioned (32 or 64-bit) address space organizations are supported.

For clustering purpose objects are allocated within segments. Segments are logically grouped into *bunches*. Each bunch has a user-level *bunch manager* implementing the policies related to persistence and distribution specific to the bunch's data: allocation, garbage collection, mapping and un-mapping, function shipping or data shipping, shared data consistency, migration, etc. Objects are referenced by maillons and SSP (stub-scion pair) chains. These mechanisms are scalable and are well adapted to support distributed garbage collection, migration and compaction.

Résumé

Nous présentons une architecture flexible qui supporte le regroupement, le stockage, le nommage, et l'accès aux objets dans un système réparti à grande échelle. Le système est logiquement divisé en *zones*; une zone est un groupe de machines ayant une organisation d'espace d'adressage homogène. Les espaces d'adressage uniformes (64-bits) de même que les espaces partitionnés (de 32 ou 64-bits) sont supportés.

Les objets sont regroupés dans des *segments*. Les segments sont logiquement groupés en *bunches*. A chaque bunch est associé un objet de niveau utilisateur: le *bunch manager*. Le bunch manager est responsable des politiques de persistance et de distribution spécifiques au contenu du bunch: stockage, ramasse-miettes, chargement et déchargement, transfert de contrôle ou tranfert de données, cohérence des données partagées, migration, etc. Les chaînes de paires souche-scion (PSS) et les maillons sont utilisés pour référencer les objets. Ces mécanismes sont adaptés aux réseaux à grande échelle et au support du ramasse-miettes réparti, de la migration et du compactage.

1 Introduction

We present an architectural model that supports clustering, storing, naming, and accessing objects in a large scale distributed system. It extends the operating system kernel by providing, among others, support for distribution, persistence, and distributed garbage collection in a large scale network.

The problem we are trying to solve is the mismatch between the wide spectrum of applications that use distributed and persistent data, and the services offered by the underlying operating system kernel. We allow programmers to modify and adapt the data management, and its sharing policies in particular, accordingly to each application's needs. Thus, we offer an extensible operating system without imposing any particular model on the applications.

The main design decisions are related with heterogeneity, user level policies for distribution and persistence, garbage collection, and objects mobility:

1. Uniform single level address spaces systems (for instance, Opal [6]) will coexist with traditional partitioned 32 and 64-bit address spaces in future local and large scale networks. On the other hand, we want to avoid as much as possible the cost of pointer swizzling [17] and one solution consists on having multiple uniform address spaces as proposed by COOL-2 [1] and Cricket[14]. Therefore, heterogeneity is due not only to different binary representations, page sizes, data alignment, etc., but also to the existence of different address space organizations. We claim that our design, in particular the use of zones, bunch managers and delayed pointer swizzling (done as latest as possible and only when it is strictly necessary) allow us to deal with all these forms of heterogeneity in a structured and flexible way.

2. The mechanisms, policies, and decisions associated with the management of persistent and distributed objects are implemented at the user level by user-level objects called bunch managers. The sharing mechanism (function or data shipping), the consistency model of shared data, the operations performed on persistent data when it is mapped or unmapped (e.g. compression, encryption, swizzling), the allocation of data, and local garbage collection, are key examples of such decisions. There is a bunch manager associated with each bunch (group of segments containing objects) that is responsible for supporting distribution and persistence taking into account the specificity of its bunch data. This concept of managers provides the operating system with hooks for supporting efficiently a large spectrum of applications.

3. Objects are referenced by maillons [8] and SSP (stub-scion pair) chains [12]. We use these mechanisms (presented in Section 3.2) because they are scalable and well adapted to support not only transparent distribution and persistence but also distributed garbage collection, objects migration and compaction.

Thus, distribution, persistence, and distributed garbage collection are supported as general and global mechanisms. Applications rely on them but will have a large amount of specificity provided by user level objects called bunch managers. Load sharing, objects mobility, clustering policies, caching strategies, consistency algorithms, are examples of such a specificity.

This paper is organized as follows. The next section presents the fundamental entities of the architectural model. Section 3 describes the functionality

associated with the bunch manager concept. Section 4 describes how persistence and distribution are handled. The paper terminates with three sections: related work, the current status of the system, and a summary.

2 Architectural Model

In this section we present the most important entities of the design: zones, segments, bunches, and bunch managers, and their relation to address spaces and protection domains.

The main components of the system are illustrated in Figure 1 which will be described in the following sections. The operating system kernel (mK) provides low-level abstractions such as virtual addresses, ports, capabilities, etc. On top of it, the BMX layer (Bunch and Managers eXecutive) supports segments, bunches, and managers.

2.1 Zones

Future networks will have a large and varied number of computing machines with distinct address space organizations (for example, uniform single-level 64-bit address space and partitioned 32-bit Unix style address spaces). In order to deal with such heterogeneity we logically divide the network in zones (simplifying the address space management). A *zone* is a logical group of machines with an homogeneous address space organization.

In Figure 1 we can see a network with 3 different zones: a uniform single level 64-bit address space (**zone 2**), a 32 and a 64-bit partitioned address space (**zone 1** and **zone 3**) as supported by the Unix operating system, for instance.

Each zone has a particular address space organization which is completely independent from other zones, i.e., there is no coordination between zones regarding the address space management. Thus, in our design, we assume the existence of an address space service per zone.

2.2 Segments

For clustering purposes objects are allocated within segments. A *segment* is a set of contiguous virtual pages with a owner and protection attributes (for example, the usual Unix read, write and execute permissions). The size of a segment is defined at creation time and remains constant until it is destroyed. Thus, the granularity of allocation and protection is the segment.

When possible, for efficiency reasons, a segment is mapped in the same address where it was last mapped avoiding pointer swizzling. However, in some circumstances (see Section 4.2.2) such a mapping may not be possible. This situation would occur more frequently when objects are mapped in a different zone from the one they were last mapped because the address space organizations are different and completely independent. In such a case, pointers must be swizzled. However, since one of our goals is to avoid the cost of swizzling we perform such an operation as late as possible, i.e., only when the segment is effectively going to be mapped.

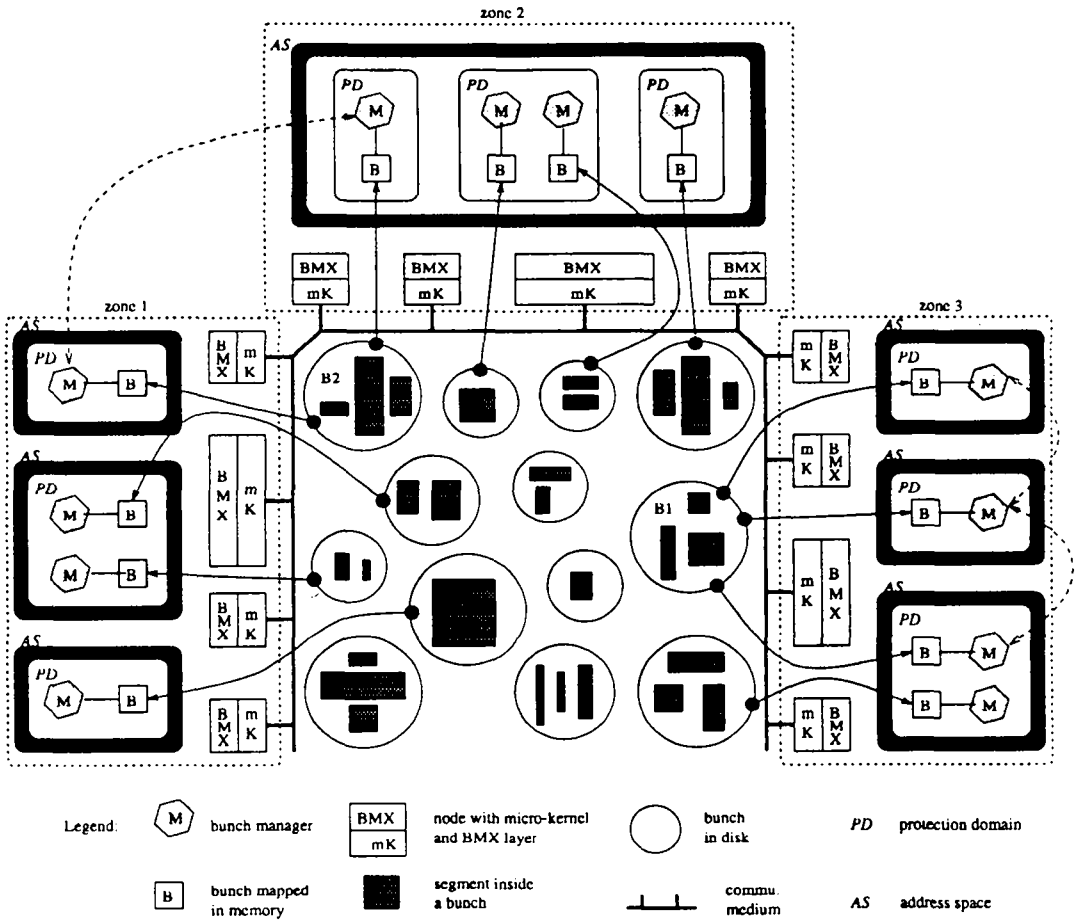


Figure 1: System main components.

Inside a zone every node shares a common address space organization but may differ on other aspects such as the byte ordering and data alignment, for example. This kind of heterogeneity may imply the modification of the segment's data when mapping. These translations (similar to the ones performed in RPC based systems) are done with the help of each enclosed object which provides the necessary methods for dealing with such heterogeneity accordingly to its language defined type. We assume that these methods are automatically generated by the compiler or some pre-processor.

2.3 Bunches

Segments are logically grouped into *bunches* for two reasons: (1) a single segment is not flexible enough for holding all applications data (for instance, segment overflow could arise), and (2) the existence of bunches provides name independence with respect to migration and compaction of objects (an object is identified relative to its enclosing bunch).

A bunch can be seen as both a file and a persistent heap. However, it is more than that because it has an associated manager responsible for its management. All segments grouped within a bunch have common management policies implemented by its bunch manager (see next section).

2.4 Bunch Managers

A *bunch manager* is a user level object associated with a bunch that is responsible for the management of the bunch's segments. In particular, the clustering policy of objects within a segment is implemented by its bunch manager.

A manager, through its methods and data structures, supports distribution, persistence, garbage collection, etc., in a heterogeneous network. Since all these mechanisms and associated policies are defined and implemented within bunch managers at the user level, there is a large amount of flexibility.

Thus the main goal of a bunch manager is to provide the necessary mechanisms and policies related to distribution and persistence taking into account the specific data within its bunch. An application will pay only the cost associated to the services provided by the bunch managers being used.

An important aspect is related with the sharing of bunches (further described in Section 4.2). The contents of bunch B1 are being shared by several threads and its consistency is maintained by its manager (in the figure there are three instances of the manager which conceptually can be seen as being only one). The same applies to bunch B2.

We will provide a library of bunch managers. Therefore, most application programmers will use already existing ones (that can be easily modified). However, new bunch managers can be programmed for specific needs.

2.5 Address Spaces and Protection Domains

Each zone has a specific address space organization. In some cases, this organization is strongly tied to the concept of protection domains. For example, in a zone with partitioned 32-bit address spaces (Unix model) a process is a protection domain. On the other hand, in a uniform 64-bit address space a protection domain may be dissociated from the address space concept (like in Opal [5], for instance).

In our design, a protection domain is defined as being simply an execution context with some zone-wide identification. The execution context comprehends those segments to which the thread or threads belonging to the protection domain may access. If an execution context also includes an address space then we have the notion of a Unix process.

In Figure 1 we can see the relation between addresses spaces and protection domains. For example, zone 2 supports a single uniform 64-bit address space where three protection domains are represented. On zones 1 and 2 a protection domain comprehends also an address space.

We keep the notions of protection domain and address space orthogonal to each other in order to deal with different models (Opal and Unix, for instance). The important point is that if a thread wants to access a bunch in a different protection domain, then its is used a trusted communication mechanism that avoids any protection violation. Such a mechanism can be a system call or a remote procedure call [2].

3 Bunch Manager Functionality

In this section we present the bunch manager concept, the reference mechanism, the object and execution model, and the bunch manager functionality related to the support of distribution, persistence, and distributed garbage collection.

3.1 The Manager Concept

A manager provides an external interface defined by its public methods (see Figure 2). This interface is a minimum common to every manager and its methods are responsible for:

- bind references, i.e, establishing a path between objects allocated inside its bunch and other objects outside it,
- providing the necessary mechanisms for supporting invocations of its internal objects (allocated inside its bunch) from external objects (allocated within some other bunch) through the path constructed when binding, and
- contribute to the distributed garbage collection by providing the information relative to incoming and outgoing references of its bunch.

The methods providing this functionality constitute the bunch manager external interface. They can be invoked on a manager from any thread (holding a reference to it) running on behalf of another manager, an application, or the underlying operating system kernel (up-call). Following sections will present in more detail the methods responsible for each of the above functions.

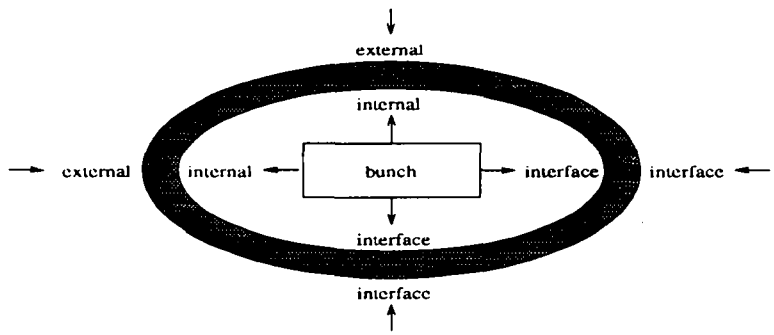


Figure 2: The manager concept.

The private methods of a manager constitute its internal interface. These methods provide the means for manipulating the data inside the corresponding bunch (to allocate memory for an object, for instance). Figure 2 illustrates a manager conceptually enclosing its bunch as every operation performed on it is controlled by its manager.

These two interfaces are provided by a kind of a *frontier* or *membrane* (shaded area in the figure) through which the bunch interacts with the exterior. This membrane can also be seen as a filter performing the necessary transformations on every data entering or leaving the enclosed bunch (e.g., swizzling pointers).

Thus, we can see a bunch manager as encapsulating its bunch. This encapsulation hides the usage of specific mechanisms and policies well adapted to the

data allocated within the bunch and by that fact provides a large amount of flexibility.

3.2 Maillons and SSP Chains

A maillon [8] is the basic reference mechanism. It is a data structure composed of a data part and a pointer to a dereference procedure (code part) that knows how to interpret the data part. The simplest maillon is formed by a direct pointer to an object and a dereference procedure. Invocations performed over such a maillon result in the invocation of the object (referenced by the data part) through the dereference procedure.

Maillons can be chained providing several levels of indirection accordingly to particular needs. For example, a remote object may be referenced by a chain of maillons in which two of them are in fact a stub and a scion (also called a stub server [4]). In this case the maillons implement a possible form of SSP chain [12].

Figure 3 illustrates the use of maillons and SSP chains within two bunches (one of them containing two segments).

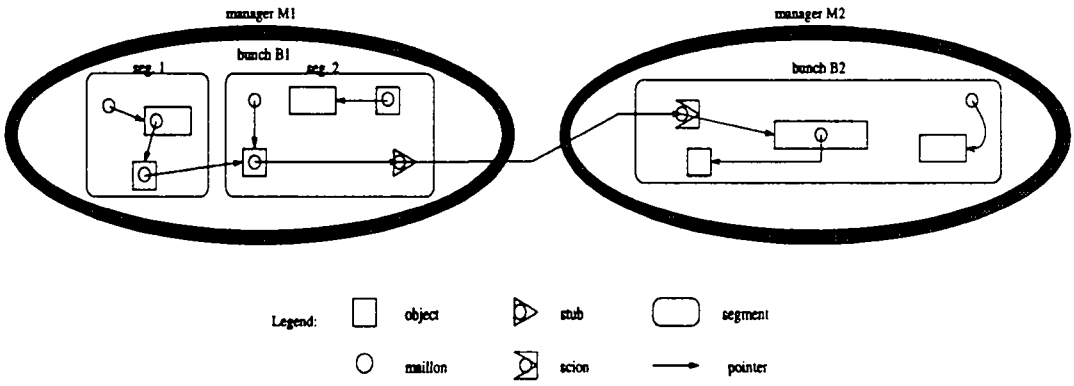


Figure 3: Objects referenced by maillons and SSP chains.

3.3 Object and Execution Model

An object is a contiguous sequence of bytes that contains no internal linked data structures and is an instance of a language type (we follow the terminology used in [13]). A type or a class is itself an object containing a group of method procedures. The granularity of identification and invocation is the object.

We assume that objects are passive and small. This means that the size of most objects is much smaller than a page. This does not preclude the existence of larger objects containing images or sounds; however, the system described is optimized for supporting smaller objects clustered in segments (for efficiency) with some user defined mechanism (implemented by their manager).

Within a bunch, objects are identified by simple maillons. Thus, invocation is a procedure call with no overhead for type- or access-checking. Across bunches, identification uses SSP chains efficiently implemented using maillons (see Figure 3).

The execution model is based on threads executing within a protection domain. A thread may be extended to another domain in order to access a bunch. This extension can be implemented with a remote procedure call preserving the protection domains.

3.4 Creating Bunches and Segments

We may create and delete a bunch using the methods `createBunch` and `deleteBunch` provided by the BMX layer. When creating a bunch, its manager is specified as a parameter along with the size and protection attributes of its initial single segment.

The creation of a bunch returns a reference to its manager. This manager is a user-level object that may be mapped in the same protection domain of the thread that has created its bunch. However, if the manager code is not trusted (or is not compatible with the local processor), the manager will be mapped in another domain (or workstation) and will be accessed by remote procedure calls.

Creating and deleting segments is done by the methods `createSeg` and `deleteSeg` also provided by the BMX layer. At creation time, the size and protection attributes are specified.

3.5 Object Allocation and Garbage Collection

Allocation of memory inside a segment of a bunch is done through the methods `allocMem` and `freeMem`. Creating an object implies not only the allocation of memory for its data but also the creation of a reference to it. References are created and deleted with the methods `createRef` and `deleteRef`. These four methods are part of the manager internal interface.

When a reference to an object is passed to outside its bunch (as an argument in a remote procedure call, for example) the private method `exportRef` is automatically invoked on its manager (`M2` in Figure 3). This export event is strongly coupled with the invocation of the private method `importRef` on the manager responsible for the bunch receiving the reference (`M1` in Figure 3). These two methods are always used together in order to maintain the coherence of the information relative to the incoming and outgoing references of a bunch. However, the garbage collector protocol is capable of dealing with incoming and outgoing information incoherencies due to network faults (e.g., `importRef` never performed due to a network partition). Therefore, the distributed garbage collector is robust and performs correctly even in presence of faults [10].

Thus, a bunch can be considered as a closed and autonomous entity with respect to object referencing and garbage collection. When the public method `gc` is invoked on a bunch manager it performs all the local operations that contribute to perform a distributed garbage collection. Basically, the invoked manager replies with the list of references to external objects (allocated within other bunches) no longer being referenced from its bunch.

Finally, a local (to the bunch) garbage collection reclaims the garbage within its segments and does not inform other managers about references to external objects no longer valid. This functionality is provided by the private method `gcLocal`.

3.6 Object References

The creation and deletion of a maillon is performed by two methods named `createMaillon` and `deleteMaillon` respectively. These methods do not belong neither to the internal nor to the external interface provided by bunch managers. They are internal to the manager membrane and are used as the result of an invocation of methods `createRef` and `deleteRef`.

A reference to a remote object (in another bunch) is done through a stub that is created within the local bunch. Within the remote bunch a scion is used. The pair stub and scion constitute a stub-scion pair (SSP). The methods `importRef` and `exportRef` invoke the methods `createStub` and `createScion` respectively which are part of the manager membrane. The connection of a stub with a scion (see Figure 3) implies a binding operation which is described in Section 3.8.

Since the methods used to create and delete maillons, stubs, and scions are not seen from outside a bunch manager, i.e., they belong neither to the internal nor to the external interface, it is possible to associate other operations to them. For example, it would be possible to send a message to some monitor (transparently to the invoker) each time a new SSP chain is created.

3.7 Object Faulting and Invocation

The invocation of an object is done by the dereference procedure of the maillon used to reference it. In the general case, if the object being invoked is allocated in a page not yet mapped in memory a fault occurs.

Faults relative to internal objects (allocated inside the bunch being used) are detected and issued by the operating system kernel with a page size granularity (page fault event). It is the manager servicing the fault that is responsible for managing the page contents and in particular the faulted object. In this sense, we can think of a segment as being a Mach memory object [18]. On the other hand, faults relative to external objects (allocated within segments not enclosed by the current bunch) are detected and issued by the dereference procedure of the maillon being used (software mechanism).

Thus, we can say that internal object faults are hardware detected (issued by the operating system kernel) and external object faults are software detected (issued by the maillon). The reason for such a design is that the information given by the page fault event is not enough for performing all the necessary operations related with the invocation of an external object. These operations are performed during the binding phase that we further describe in Section 3.8 and handles all the aspects related with type- and access-checking, sharing mechanisms and policies, etc.

A fault issued by the kernel results in the invocation of the method `bmxHandler` which is part of the BMX layer. A software fault consists in the execution of the dereference procedure associated with the maillon. In both cases (illustrated in Figure 4) the method `bind`, of the manager whose bunch originated the fault, is invoked (from the method `bmxHandler` (1) or from the dereference procedure (1')).

Thus, the manager of the bunch from which the fault originated (manager `M1`) is responsible for servicing it by performing the operations that allow the continuation of the faulting thread. In particular, it will invoke the method

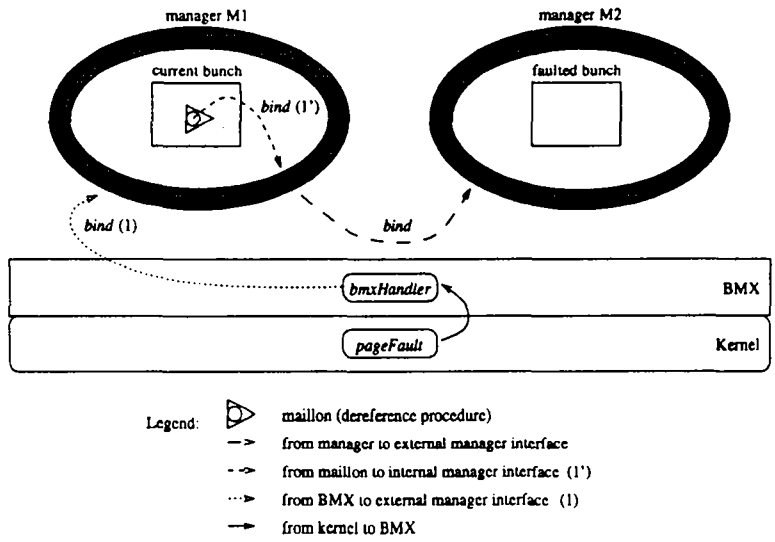


Figure 4: Invocation of the method bind to handle a fault.

bind on the manager of the faulted bunch (manager M2). Both managers will negotiate the access to the faulted data in order to allow the continuation of the faulting thread. If several threads are using the same bunch, the negotiation mentioned above will include the decision on how to share an object or a segment. In fact, this is one of the tasks performed by the binding phase described next.

3.8 Binding

The ultimate purpose of the binding operation is to set up a path between a source and a target object. Such a path can be achieved by connecting a stub and a scion (illustrated in Figure 3), or by bringing the target object close to the source (loading an object on memory from disk, for instance).

The method bind may be invoked on a manager in several circumstances: (1) as an up-call from the underlying BMX layer in order to service an internal object fault, (2) from the dereference procedure of a maillon to service an external object fault, (3) explicitly from the application as a way of performing objects *pre-fetching*.

The most important operations done when binding are listed below (not all of them are always performed):

- reference redirection: the reference is redirected to the target.
- access checking: the invoker object must be authorized to perform the requested access.
- type checking: the type of the object being accessed must conform with the type assumed by the invoker object.
- sharing decision: decide on the invocation mechanism, function shipping by remote procedure call (RPC) or, data shipping via distributed shared memory (DSM).

- dynamic loading: the faulted object or a stub is brought from disk and loaded in memory in the case of data shipping or function shipping respectively.
- dynamic linking of code: the loading of an object may imply the dynamic linkage of its code.
- communication protocol: the transport and presentation protocols must be chosen (in the case of function shipping).
- consistency semantics: what kind of consistency should be maintained (in the case of data shipping).

All these aspects are part of the binding operation which is a negotiation between the running thread (executing the `bind` method of `M1` in Figure 4) and the manager of the bunch to be accessed or shared (faulted bunch managed by `M2`).

4 Persistence and Distribution

This section presents some of the items of the binding phase with relevance to the support for persistence and distribution. The other points are out of the scope of this paper and some of them are discussed in another article [11].

When a remote object (in another bunch) is accessed, the first operations that have to be performed are the access- and type-checking. When the object is of a previously unknown type its code is mapped in memory and dynamically linked. Since its type is just another object, a new type is faulted in when first called. In the following sections we assume that all these manipulations are performed as the first steps of the binding phase.

4.1 Persistence

Objects and segments are created volatile. Segments of a bunch, and the bunch itself, will become persistent only if at least one of the enclosed objects is made persistent. The internal interface of a manager supporting persistent objects provides methods to transform a volatile object into a persistent one associating it with a symbolic name. In particular, there are methods to register an object within the name service and to obtain a reference given its symbolic name.

Consider the case when a thread tries to invoke a persistent object (allocated within some other bunch) through a reference received from the name service (an SSP chain containing a maillon not yet bound). The dereference procedure of the maillon will invoke the `bind` method on its manager. This method will negotiate with the manager of the persistent object in order to access it (binding phase).

A possible result from the binding consists on mapping the persistent object (and the enclosing segment) in the protection domain of the client thread and redirecting the maillon. Once the binding is complete the dereference procedure will execute the invocation. However, it can happen that the segment containing the persistent object can not be mapped in the protection domain of the client thread because it belongs to a bunch of a different domain. In this situation the binding phase could, for instance, create a stub bound to a scion through which remote invocations would flow.

4.2 Distribution

The most important point related to distribution during the binding phase is the sharing decision. It consists on deciding what kind of mechanism and policy should be used to invoke the shared objects.

There are two basic possibilities for performing the invocation: function shipping by remote procedure call (RPC), or data shipping via distributed shared memory (DSM). In the first case, the stub references a scion in the remote bunch which points to the object. The invocation is performed through this link. In the second case, the object is mapped by both threads and its consistency is maintained by its bunch manager (recall Figure 1 where bunch B1 is being shared).

A large number of factors (page and object size, protection domains, number and type of accesses, etc.) influence the sharing decision and the fundamental point is that there is not a universal solution that performs efficiently for every situation. Some basic decisions can be implemented automatically (e.g., using RPC or DSM), based for instance, on the number of consecutive invocations performed the last time an object was accessed. However, we think that such automatic decisions impose a serious performance cost and are successful only in a small number of simple situations.

Thus, we claim that it must be possible for the application to give some compile- or run-time hints to the supporting system about the most convenient invocation mechanism and sharing policy. These hints may be constructed with some compile-time tool (integrated with the programming environment) or by the application programmer himself.

These hints are the best way for improving the performance of an application since the programmer (or some high-level tool) has greater knowledge about the application structure than the underlying support system. In particular, the programmer knows the type of each object and all the synchronization constraints specific to the application.

Hints can be general, i.e., common to a group of objects and/or valid for a certain amount of time, or specific to each object. In the first case, hints are transmitted to the bunch manager by invoking its `bind` method or some other configuration method. In the second case, the binding phase is augmented with the invocation of some particular methods of the object itself.

Finally, an application programmer may even modify the already existent `bind` method in order to adapt it to his specific requirements.

4.2.1 Function Shipping

Between zones or between different protection domains the most usual mechanism supporting sharing will be the remote procedure call. Remote procedure calls are handled by SSP chains which are managed accordingly to a set of protocols: (1) the Transport Protocol that specifies which messages are accepted, (2) the Presentation Protocol says how to marshall a reference into, and unmarshal it from, a message, (3) the Invocation Protocol details how, when invoking an object, it is located and any indirect SSP chain is short-cut, and (4) the Cleanup Protocol eliminates scions associated with garbage remote references. More details about these protocols can be found in another article [12].

4.2.2 Data Shipping

The existence of several zones with different address space organizations introduces some complexity in the sharing decision, mainly when distributed shared memory is used. This arises from the fact that the creation of segments depends on the address space organization. For example, in a zone with a traditional Unix partitioned address space organization, each process is free to create a segment in any address (using `malloc` or `mmap`, for instance) that does not conflict with operating system constraints without performing any zone-wide agreement with other processes. On the contrary, in a uniform address space as implemented by Opal, the creation of a segment must obey a zone-wide criteria of segments allocation in order to avoid overlapping of addresses.

Given this heterogeneity of allocation criteria, the problem that arises is related with the sharing of segments created in different zones because it may not be possible to map a segment always in the same address. As a general rule, a segment will be mapped at the same address where it was mapped the previous time, but the manager can be instructed to remap it at a different address (by swizzling pointers).

It could be argued that sharing between heterogeneous zones would be easily done using always remote procedure calls avoiding not only the mentioned mapping difficulties but also the problems due to the existence of 32 and 64 bits processors (e.g. converting basic data types between 32 and 64 bits). However, it might be useful for applications to use distributed shared memory even between heterogeneous zones. Such decision will be mainly based on the size of the shared data along with the computation performed on it and the speed ratio of the concerned processors.

Another important point is related with the consistency of the shared data. The manager of the data being shared is responsible for keeping its consistency. Different consistency models (e.g. entry consistency [3]) can be provided by different bunch managers accordingly to the specificity of the data allocated within the corresponding bunch.

A bunch manager can even adapt its consistency algorithm to the particular synchronization needs of an application. For example, when performing the binding phase for accessing an object which is the root of a shared tree (allocated in some remote bunch) we may have at least two situations: (1) only the root object is effectively going to be accessed, (2) all the tree will be accessed.

In the first case we may use function shipping for invoking the object and the binding phase will automatically lock it avoiding a second invocation. In the second case the binding phase will decide to use data shipping by mapping all the segment (or segments) containing the tree close to the client thread and will invoke a special method on the root object in order to lock all the tree. All these optimizations are easily supported by the concept of bunch managers and the possibility of providing them with user-level hints.

5 Related Work

Many systems supporting persistence and/or distribution have been designed and built in the last years. In this section we present only a few and compare them with our design.

5.1 Opal

Opal [6] supports a single uniform virtual address space spanning a local area network of homogeneous processors including data on long-term storage. Direct virtual memory pointers are used as distributed and persistent identifiers facilitating sharing. All data is stored in non-overlapping virtual segments.

In comparison with our design, the most important differences are: (1) we envisage a network with several heterogeneous machines with different address space organizations, (2) we group segments into bunches which allow us to provide independence to segment overflow and address changes, (3) bunch managers provide a way to map user-defined semantics (policy decisions) on system mechanisms, (4) we support persistence, distribution, and distributed garbage collection as general and global mechanisms, and (5) objects are referenced by maillons and SSP chains which are not as fast as direct pointers but are much more flexible.

5.2 COOL-2

COOL-2 (Chorus Object Oriented Layer) [1] is a layer built above the Chorus micro-kernel [7] that extends its abstractions with support for object oriented systems. It is provided a generic support for clusters of objects in a distributed virtual memory model. The lowest layer of the system supports only clusters and the upper layers support objects. Our design was inspired by COOL-2 but is simpler and more flexible due to the fundamental concept of bunch managers that provides a larger amount of flexibility. Another difference is that we explicitly consider the existence of 64-bit microprocessors and therefore the viability of a single uniform 64-bit address space.

5.3 IK

IK [9, 16] is an object oriented platform that simplifies the construction of distributed applications handling persistent data. The most important differences are the following. In IK, objects are identified by globally unique identifiers (making the system usable only in small networks) and are always shared using remote procedure calls. The platform supports heterogeneous workstations but the address space model is homogeneous (process model).

5.4 Storage Abstractions

Soulard [15] describes the design of a flexible storage system. A fundamental goal is to reuse basic storage components and the ability to tailor the system to specific needs. To achieve such goals two entities are provided, the container and the cluster. The container defines how data is stored and the cluster manages the sharing of data. Thus, our system can be seen as implementing a particular family of clusters supporting complex distribution policies in networks with several address space organizations and assuming the existence of containers providing the support for storing persistent objects.

6 Status

The overall design of the system is finished. We already have some components available like the maillons and SSP chains supporting remote procedure call and distributed garbage collection. A complete prototype will be implemented on a network comprising DEC Alpha and Sun workstations running OSF/1 and Sun OS respectively. We also envisage the usage of a micro-kernel such as Mach or Chorus.

We intend to take advantage of the huge address space provided by 64-bits microprocessors by implementing our system on a single uniform address space shared by several workstations. In such a zone we will implement our data structures (maillons and SSP chains, for example) much more efficiently.

7 Summary

We presented an architectural model of a system supporting distributed applications with persistent objects and distributed garbage collection. The system deals with several forms of heterogeneity and in particular with the existence of different address space organizations taking into account uniform 64-bit address spaces.

Objects are allocated within segments grouped in bunches and are referenced by maillons and SSP chains. These identification mechanisms are well adapted to large networks since they are easily scalable.

The manager concept is a fundamental aspect of the architectural model that is implemented by user level objects called bunch managers. Managers are responsible for the management of the enclosed bunches by implementing the policies and mechanisms related with distribution, persistence, and distributed garbage collection. An application will pay only the cost for the services provided by the bunch managers being used.

8 Acknowledgements

We would like to thank Sape Mullender for his comments on an initial version of the design and, Hank Levy, Povl Koch, Hervé Soulard and, Laurent Amsaleg for their constructive criticism. Finally we thank Julien Maisonneuve and David Plainfossé for their work on maillons and SSP chains.

References

[1] Paulo Amaral, Rodger Lea, and Christian Jacquemot. Implementing a modular object oriented operating system on top of Chorus. In *OpenForum*, pages 193–204, Utrecht (NL), November 1992.

[2] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 102–113, Litchfield Park AZ USA, December 1989. ACM.

[3] Brian N. Bershad and Matthew J. Zekauskas. The Midway distributed shared memory system. In *Proceedings of the COMPCON'93 Conference*, pages 528–537, February 1993.

[4] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Programming Languages and Systems*, 2(1), February 1984.

[5] Jeff Chase, Hank Levy, Miche Baker-Karvey, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. In *Third Workshop on Workstation Operating Systems*, Key Biscaine, Florida (USA), April 1992. ACM.

[6] Jeffrey S. Chase, Henry M. Levy, Miche Baker-Harvey, and Edward D. Lazowska. Lightweight shared objects in a 64-bit operating system. In *Conf. on Object-Oriented Programming Systems, Languages, and Applications*, volume 27 of *SIGPLAN Notices*, pages 397–413, Vancouver (Canada), October 1992. ACM Press.

[7] Michel Gien and Lori Grob. Micro-kernel based operating systems: Moving UNIX onto modern system architectures. In *Proceedings of the UniForum'92 Conference*, San Francisco, USA, January 1991. Also available as TR-91-81, Chorus Systèmes, Saint-Quentin-en-Yvelines (France).

[8] Julien Maisonneuve, Marc Shapiro, and Pierre Collet. Implementing references as chains of links. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 236–243, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[9] José Alves Marques and Paulo Guedes. Extending the operating system to support an object-oriented environment. *OOPSLA' 89 Conference Proceedings*, 24(10):113–122, October 1989.

[10] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Tenth Symp. on Reliable Distributed Sytems*, Pisa (Italy), October 1991.

[11] Marc Shapiro. Identifying objects in a large-scale distributed system. Note technique SOR–138, Projet SOR, INRIA, Rocquencourt (France), June 1993.

[12] Marc Shapiro, Peter Dickman, and David Plainfossé. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), nov 1992. Also available as Broadcast Technical Report #1.

[13] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In Stephen Cook, editor, *ECOOP'89, Proc. of the Third European Conf. on Object-Oriented Programming*, British Computer Society Workshop Series, pages 191–204, Nottingham (GB), July 1989. The British Computer Society, Cambridge University Society.

[14] Eugene Shekita and Michael Swilling. Cricket: a mapped, persistent object store. In *Proceedings of the 4th Workshop on Persistent Object Systems (POS)*, pages 89–102. Morgan-Kaufman, September 1990.

[15] Hervé Soulard and Mesaac Makpangou. A generic FO-structured framework for persistence support in distributed settings. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 57–65, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[16] Pedro Sousa, Manuel Sequeira, André Zúquete, Paulo Ferreira, Cristina Lopes, José Pereira, Paulo Guedes, and José Alves Marques. Distribution and persistence in the IK platform: Overview and evaluation. *To appear in Computing Systems (Fall 1993)*, 6(4), 1993.

[17] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *1992 Int. Workshop on Object Orientation and Operating Systems*, pages 364–377, Dourdan (France), October 1992. IEEE Comp. Society, IEEE Comp. Society Press.

[18] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, Austin TX (USA), November 1987. ACM.

Contents

1 Introduction **3**

2 Architectural Model **4**

2.1 Zones 4

2.2 Segments 4

2.3 Bunches 5

2.4 Bunch Managers 6

2.5 Address Spaces and Protection Domains 6

3 Bunch Manager Functionality **7**

3.1 The Manager Concept 7

3.2 Maillons and SSP Chains 8

3.3 Object and Execution Model 8

3.4 Creating Bunches and Segments 9

3.5 Object Allocation and Garbage Collection 9

3.6 Object References 10

3.7 Object Faulting and Invocation 10

3.8 Binding 11

4 Persistence and Distribution **12**

4.1 Persistence 12

4.2 Distribution 13

4.2.1 Function Shipping 13

4.2.2 Data Shipping 14

5 Related Work **14**

5.1 Opal 15

5.2 COOL-2 15

5.3 IK 15

5.4 Storage Abstractions 15

6 Status **16**

7 Summary **16**

8 Acknowledgements **16**

Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399