Persistent Shared Object Support in the Guide System: Evaluation & Related Work

Daniel Hagimont *, P.-Y. Chevalier †, A. Freyssinet,
S. Krakowiak, S. Lacourte, J. Mossière, X. Rousset de Pina

Bull-IMAG/Systèmes, 2 av. de Vignate, 38610 Gières - France
Internet: Daniel.Hagimont@imag.fr

Abstract

The purpose of the Guide project is to explore the use of shared objects for communication in a distributed system, especially for applications that require cooperative work. Since 1986, two prototypes have been implemented respectively on top of Unix (Guide-1) and Mach 3.0 (Guide-2). They have been used for the development of distributed cooperative applications, allowing us to validate or reject many design choices in the system.

This paper gathers the lessons learned from our experience and compares the basic design choices with those in other distributed object-oriented systems. The lessons may be summarized as follows. This system layer must provide a generic interface for the support of several object-oriented languages. It must manage fine grained objects and enforce protection between objects and processes. These requirements can be achieved with an acceptable trade-off between protection and efficiency.

*This author is currently a post-doctoral invitee at the University of British Columbia, in the Department of Computer Science, 2366 MainMall, Vancouver, B.C. Canada V6T 1Z4. This position is funded by INRIA (Institut National de Recherche en Informatique et Automatique), Domaine de Voluceau - Rocquencourt, B.P. 105, Le Chesnay Cedex, France.

†This author is currently member of the staff of the European Computer-Industry Research Center (ECRC), Arabellastrasse 17, D-81925 Munich, Germany.

OOPSLA 94- 10/94 Portland, Oregon USA

1 Introduction

Support for cooperative distributed applications is an important direction of computer systems research, involving developments in operating systems as well as in programming languages and databases. One emerging model for the support of cooperative distributed applications is that of a distributed shared universe organized as a set of objects. In this paper, we report on our experience in designing, implementing, and using a system to support such a model.

The main requirements for the class of applications that we consider may be summarized as follows. The system should provide access to shared information, with a fine granularity. Sharing may be concurrent, which involves fine-grained access synchronization and selective protection. The system should provide location-transparent long term storage. In addition, we are interested in systems that provide support for programming languages; the primary requirement here is related to persistence.

We have selected an information structuring

model based on shared objects. Objects are passive, i.e. active agents (processes or threads) are defined independently from objects. Every object is potentially persistent, i.e. its life span is unrelated to that of the process or application in which it was created. Every resource or abstract entity is represented by an object, and all communication between processes takes place through shared objects. Variations of this model have been explored in several projects (e.g. Emerald [Black86]). Its main benefits are abstraction, modularity and reusability.

Our research effort has been done in two phases. We started by building a prototype of an object support system (Guide-1), based on Unix[1], and tuned to the needs of one specific language, also designed by our group. The experience gained from the use of this system was used to design a generic object support subsystem (Guide-2), developed on the Mach 3.0 micro-kernel [Acetta86].

The contributions of this paper are the following:

- we present the implementation principles of object support in the Guide system,
- we compare our approach with related work,
- we gather the experience and the lessons learned.

The next section presents an overview of the Guide project, including the motivations for each prototype implementation. Section 3 describes the design, implementation and lessons learned of the Guide sytem. It focusses on the object model, object management, object naming, object binding and object protection. We summarize and conclude in section 4.

2 Overview of the Guide project

The Guide project started in 1986 with the goal to experiment with the support of object-oriented languages at the system level for the development of cooperative distributed applications.

As a first experience in shared object support, we designed and implemented the Guide-1 prototype. This work was carried out from 1986 to 1990. Two basic design choices underlie the implementation of the Guide-1 prototype:

- First, the system was tuned to the needs of a single language. Defining our own language, rather than extending an existing language with the required facilities for concurrency, persistence, and distribution, provided the freedom of design required in an exploratory project. The Guide language has strong, static typing; this allows protection to be based on checking by a compiler. Types (interface descriptions) are distinct from classes (instance generators); a type may be implemented by several different classes. More details on the language can be found in [Krakowiak90].
- Second, the Unix system was used for the implementation of this prototype. Actually, Unix was not chosen for its features regarding the Guide requirements, but rather for its adequacy for rapidly developing a platform for our experiments.

After four years of efforts (1986 to 1990), a compiler for the Guide object-oriented language and a runtime for this compiler were developed. This runtime provided object sharing and object persistence with distribution of naming, storage and execution.

This platform allowed the development of full scale applications such as a distributed diary, a system for document circulation [Cahill93a], a directory service, and a distributed cooperative editor [Decouchant93]. It was also used to address some other problems such as garbage collection [Nguyen91] or concurrent application debugging [Jamrozik93].

However, Unix was found to be inadequate for the support of the Guide system, especially for the management of object sharing. Moreover, some

[1] Unix is a trademark of UNIX Systems Laboratories, Inc.

critical problems were poorly or not addressed in this first implementation:

- Method call was not efficient in Guide-1.
- Even if the Guide language was highly appreciated by its users, we felt the necessity to support the most used object-oriented language, i.e. C++.
- In Guide-1, object encapsulation was enforced by the Guide compiler and based on strong typing. If the system is intended to support untrusted compilers (such as a C++ one), then the system must be in charge of object isolation enforcement.
- A system that allows object sharing must provide mechanisms for access control. Such facilities were not provided in the Guide-1 prototype.

Therefore, we began in the fall of 1990 the design of a new prototype (called Guide-2) on the Mach micro-kernel, based on the experience gained from the first implementation.

At this time, this second implementation (1990 to 1993) has been completed. The Guide-2 system is running on the Mach3.0 micro-kernel[2] on a network of 486 PCs. The runtime implemented by this kernel provides the support required both for the Guide language compiler and for a C++ compiler in which objects are persistent and may be shared between the execution structures. Object addressing is far more efficient; protection mechanisms have been integrated in the kernel and allow the enforcement of users and object isolation and the development of protected applications.

We are currently working on porting the application set that was developed on the Guide-1 prototype and also on the completion of the development environment.

[2]We benefited from the support of the OSF Research Institute of Grenoble.

3 Design, implementation and lessons learned

We now present the design and implementation of persistent shared object support in Guide. This presentation is composed of several subsections respectively devoted to different aspects of the system. For each of these subsections, the first part describes the Guide design (of both prototypes where applicable) and the second part provides a comparison with similar systems and the lessons learned from our experience.

3.1 The Guide model

3.1.1 Design

The Guide system is based on shared, passive objects. The choice of a passive object model was motivated by measurements of object sizes, which show that applications tend to use a large number of small objects. We considered that active object models would be more appropriate for "heavy-weight" objects such as servers.

The execution model is organized in tasks (virtual address spaces in which the objects are mapped); concurrent activities run inside tasks. Both tasks and activities may be distributed. This model was chosen to investigate the paradigm of shared objects as a single means for communication: thus activities (within a task or between tasks) interact through shared objects; synchronization constraints [Decouchant91], expressed in the language and implemented by the run-time system, are associated with shared objets.

Objects are persistent, i.e. their lifetime is independent from that of the task in which they are created. This decision was essentially motivated by uniformity (managing only one kind of object).

At the model level, the main difference between our two prototypes is in the interface between the system and the compiler(s). Guide-1 defined a single-language virtual machine. A major goal of the design of Guide-2 was to provide a generic virtual machine allowing the support of different languages satisfying a minimal set of assumptions.

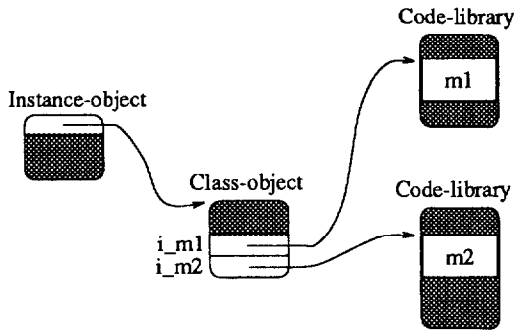

Figure 1: The generic object model

This virtual machine (described in [Freyssinet91]) provides a generic object model which includes the basic abstractions for building more complex object models. The practical goal was to support the Guide language and a persistent extension of C++. The model defines three basic abstractions: instance-objects, class-objects, and code-libraries. The corresponding entities are persistent and named by universal system references (more details in section 3.3). Figure 1 shows the organization of these entities.

Class-objects and instance-objects are defined separately to enforce modularity; the system knows about the link between an instance object and its class object. An instance-object can only be accessed using the methods defined in its class-object. On the other hand, the system has no knowledge of inheritance, because of the wide diversity of inheritance models. Inheritance is managed by the run-time system of the language. Thus the system does not manage relationships between class-objects. The code of the methods involved in class definitions is stored in code-libraries. A class-object is actually a descriptor for the class: it contains references to the code-libraries which include the code of the class methods. Objects may contain references to other objects. A code-library may contain a reference to a procedure in another code-library.

3.1.2 Lessons learned and related work

Shared persistent objects

Shared, transparently distributed objects, with high level language support, were considered an extremely useful tool by the programmers of distributed applications, in contrast to explicit messages. In addition to the benefits specific to the language (strong typing, conformity, multiple implementations of a type, etc), the main advantages mentioned were: the higher degree of abstraction for the expression of distribution (an application developed on a single node could be ported to a network without change); the ability to build large structures with embedded object references and the ability to share substructures ; the separate expression of synchronization constraints for shared objects; the implicit management of persistence. The lessons are relevant for many similar systems (e.g Emerald [Black86], Orca [Bal87]).

Generic object model

The Guide-2 system provides a generic object model for the support of several object-oriented programming languages (OOPLs). Several projects attempted to provide such a generic platform.

Some earlier systems, such as Clouds [Dasgupta90] or COOL v1 [Habert90], tried to directly map OOPLs on abstractions provided by some system kernels, essentially address spaces or segments. Since OOPLs deal with fine-grained objects, these prototypes either provided two kinds of objects, local objects managed by the compilers and global system objects (global naming not being supported for local objects), or suffered from poor performance because of the mismatch between system provided abstractions and language required ones.

More recent prototypes, such as Amadeus [Cahill93b] or COOL v2 [Lea93], provide fine-grained object support directly at the system level. But in order to avoid modifying the supported compilers and in particular to keep language specific invocation schemes, they both use an upcall mechanism for the system to request information about

object management at the language level. Upcalls are notably used to locate external references in objects, and to replace these references by virtual addresses used in most of the languages. However, in most of the cases, the supported compilers have been modified (or a preprocessor added) in order to generate class specific upcall functions.

While the Guide project shares the same objectives, it implements the generic platform differently. The upcall-based system layers which aim at avoiding compiler modifications do not actually reach this goal. Since compiler modifications are inevitable, it seemed preferable to focus on the adequacy and the efficiency of the provided mechanisms. Therefore, we decided to design a kernel that efficiently provides mechanisms for sharing, persistence and protection, without the constraint to keep compilers unchanged. The Guide-2 system implements a virtual machine that provides a generic object model. We believe that this generic object model is a common denominator of all the potentially supported languages. A new invocation scheme has been designed; we have modified the Guide-1 compiler and implemented a C++ preprocessor that generates application code for this virtual machine.

3.2 Object management

3.2.1 Design

The most important problem in object management is to provide efficient support for sharing while ensuring protection.

In Guide-1

In the Guide-1 prototype [Balter91] on Unix, each activity was implemented by a Unix process. Activites (Unix processes) on one node were sharing objects by sharing a large memory region in which objects were loaded on demand. An object was loaded on only one node at a time and remotely-loaded object invocations were performed with remote procedure calls.

Therefore, the address space of a task shared between its activities had no concrete implementation. On each node, all the running activities were sharing the same object space in which objects were loaded. Object isolation (encapsulation) relied on the safety of the code produced by the Guide compiler.

In Guide-2

In the second prototype on top of Mach 3.0, Mach tasks (also called context in our jargon) and threads were used to implement our tasks. The main motivation was to enforce task and object isolation at the system level:

- An error in a task should not generate an error in another task with which it does not share any object.
- In one task, an error in an object should not be able to corrupt any object shared by the task. In particular, we wanted to enforce isolation between object owners.

First, since we wanted to enforce isolation between tasks, we rejected the solutions where tasks share entire address spaces, and especially solutions based on context sharing between tasks. Therefore, in the current version, a task is a set of contexts and object sharing is performed with the Mach mapping mechanism (without isolation enforcement, it may be considered that there is one context per node where the task is represented).

Second, in order to enforce isolation between objects, we decided that objects of different owners must be mapped in different contexts. Thus, a task may be composed of several contexts on one node, each of them associated to a different object owner. When an activity spreads from an object owned by user X to an object owned by Y, it must execute a cross-context invocation, which is interpreted by the system. Thus, an addressing error in a method of an object can only affect objects having the same owner. Although this management does not provide complete object isolation, we think that such an isolation on a per-owner basis is a good trade-off.

The last basic choice for object management relates to object clustering. Our experience with Guide-1 showed that most of the Guide passive objects are small (i.e. less than 300 bytes). Using

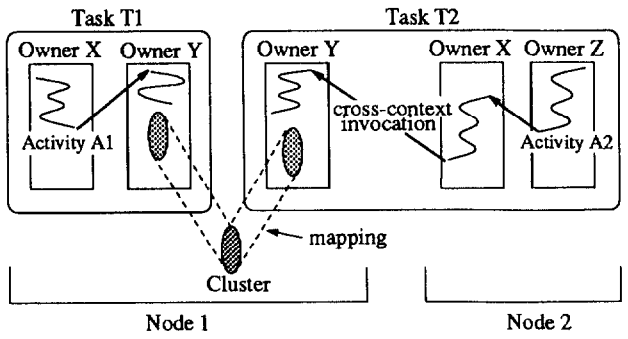

Figure 2: Object management in Guide-2

objects as units of sharing would mean supporting the cost of a mapping for each object binding. We therefore decided to use an object clustering scheme. A cluster is a set of (logically related) objects and the unit of mapping is the cluster. A cluster is mapped in a task when one of its objects is first used in that address space. If clustering can be used at a higher level to group logically related objects, then the mapping of an object implies the mapping of a working set of the object. Notice that since the cluster is the unit of mapping in contexts, all the objects stored in a cluster must belong to the same owner.

A tag associated with each cluster indicates whether the cluster can be mapped on different machines. If an activity needs to access a remotely mapped cluster that can only be shared on one node, the requesting activity will extend to that node and map the cluster locally. The extension of the activity may require the creation of a new context on that node.

Figure 2 illustrates object management in the Guide-2 prototype.

3.2.2 Lessons learned and related work

User and Object isolation

The Guide-1 prototype may be compared to the Emerald system [Black86] in the sense that they both implement a language machine and both systems achieve node-wide sharing of an object space. In both cases, the system trusts the compiler used for application development.

In the Guide-2 design, the support of unsafe languages required protection mechanisms at the system level. Therefore, the Guide-2 system maintains a strong isolation between execution structures and objects. Its design may be compared to issues employed in similar projects.

Systems such as Argus [Liskov85] or Clouds [Dasgupta90] enforce isolation for coarse-grained objects, each context being associated with a unique object. Since local objects are managed at the language level, we may compare their objects to our clusters. As previously noted (section 3.1.2), the major difference is that Guide-2 manages a global naming for fine-grained objects. Moreover, Guide-2 manages clusters that may be of various sizes (even small, i.e few pages) and these clusters may be mapped in the same context (if they belong to the same owner). It ensures a more efficient object invocation between objects from differents clusters when mapped in the same context, since it avoids a systematic cross-context invocation.

In some other systems (e.g. Amadeus [Cahill93b]), contexts are used as clusters servers: a cluster is dynamically mapped in the first context which requests it and every further method call on objects stored in this cluster occurs in this context. We rejected this solution in order to enforce our task isolation and also for the following reasons:

- The management of isolated tasks (in which clusters are dynamically mapped) allows an easier identification of the state of the running application. In a server based organization, application threads may be running in contexts (servers) that are shared between applications.

  In particular, this task isolation has simplified user authentication in the system (since a context is only running for one task on the account of one user) and the management of user level I/O.

- It also allowed us to perform selective binding according the user's rights on the called

object. At binding time, an access path in the current context is build according protection informations, and this path depends on the user associated with the task the context belongs to (detailed in section 3.5).

Object clustering

Because of the mismatch between abstractions provided by system kernels such as Unix, Mach or Chorus and the OOPLs requirements, object clustering is inevitable and is implemented in most of the systems. However, clustering is more or less easy to manage according the kernel provided features.

In the Guide-1 system (on Unix), clustering persistent objects would have meant loading a possibly large set of objects in a Unix shared memory segment; since this would have been wasteful, objects were loaded on demand.

In the Guide-2 design, Mach 3.0 provides the ability to handle page faults in a task called the "external pager". Therefore, it is possible to map a cluster in a context and to load a page from the persistent storage when requested from the external pager [Balter93]. The external pager facility is one of the more valuable Mach features: it allows a clear separation between the object as a unit of addressing, the cluster as a unit of sharing and the page as a unit of I/O.

3.3 Naming and locating objects

3.3.1 Design

The design of a naming scheme must take two constraints into account:

- The size of the object identifiers. Object identifiers must provide the ability to name a large number of objects, but the size of these identifiers must be kept small in order to reduce disk occupation and CPU time.
- The purpose of migration. In our case, object migration is principally used to gather objects in clusters and to gain on futher accesses. An object migration must shorten the object location (i.e. make it faster).

In Guide-1

In Guide-1, the object space in secondary storage is divided into *containers*. Objects are named by 32 bit object identifiers (*Oids*) allocated at creation time; these identifiers contain the creation container identifier. A System Reference (*SysRef*) that points to an object is a 64 bit variable, the first 32 bits being the *Oid* of the pointed object and the last 32 bits being an hint for the current location of the object.

Therefore, *SysRef* comparisons use only the *Oid* field of the *SysRef* without the need to access the object, and object location from a *SysRef* is immediate if the hint is up-to-date. If not, an up-to-date hint that is always stored in the creation container is used and the hint in the *SysRef* is updated.

This scheme is good for migration in the sense that it will always converge to the cheaper location cost (when the hint in the *SysRef* is up-to-date). However, its major drawback is that it only allows the naming of objects on 32 bits with some 64 bit system references. The support of users that developed large scale applications showed us that 32 bit *Oids* were not enough.

In Guide-2

In Guide-2, we wanted to use 64 bit object identifiers and to manage object migration without extending system references to 128 bits.

Similar to the previous prototype, an object name is allocated at creation time and contains the initial location of the object. Assuming that object migration is infrequent, an object is often found in its creation cluster.

In order to allow object migration, we use a technique based on migration catalogs and forwarders to locate migrated objects. An object that migrates from its creation cluster is registered in a catalog associated with its new location cluster and a forwarder is left in its creation cluster. The catalog is part of the data of the cluster.

Suppose an object *O2* is invoked and *O2* has to be located. If the creation cluster of *O2* (given by the name of *O2*) is already mapped, then we try to locate *O2* in it (this is the default case); if *O2* migrated to another cluster, a forwarder will be

found in the creation cluster. If the creation cluster of *O2* is not already mapped, then the catalogs of the already mapped clusters in the current context are used to check if *O2* migrated to one of these clusters. If not, the creation cluster of object *O2* is mapped and *O2* is searched.

With this strategy, we never remap a cluster for a migrated object that resides in an already mapped cluster. Moreover, we only pay for a lookup in the migration catalogs when a cluster mapping is required. Remember that the cost of a map operation is much greater than the cost of a lookup in the catalogs of the currently mapped clusters.

3.3.2 Lessons learned and related work

The main motivations for the design of our naming scheme were the naming of a large object space, the efficiency of the location process, a migration function which speeds up object location when objects are grouped in clusters, and to keep the size of object identifiers small.

The solution we implemented is based on 64 bit identifiers; it reduces the location cost for a migrated object when the object is already mapped, and it does not degrade location performance for objects that do not migrate, since there is overhead only when a cluster has to be mapped (the location of an already mapped object that did not migrate is unchanged). However, the solution needs to keep a forwarder to the actual location of a migrated object in its creation cluster. This means that the location process of an object is highly dependent on the object's creation cluster and its availability. This scheme should be improved by the management of reliable migration servers that register migrations for objects that need this high availability.

This solution may be compared to three classical ones:

- Forwarders. The use of simple forwarders deals with migration, but it never reduces the cost of locating migrated objects.

- Location independent naming. These solutions are often based on location servers. The location cost is then constant whenever the object migrated or not, but it is a detriment to objects that never migrate.

- Content dependent identifiers. With this solution, each object identifier (contained in an object state) is local to the current cluster, i.e. it always refers to an object in the same cluster. A reference to an object located in another cluster is a local identifier to a special object which is a forwarder (it contains the identifier of the cluster and the local identifier of the referenced object in this cluster). An object migration is then very simple: the object state is copied, the initial state is replaced by a forwarder and references to this object in the destination cluster (that were through a forwarder) are replaced by the new local identifier of the object. The system can shortcut chains of forwarders when they exist. Therefore, the gathering of related objects in the same cluster reduces the location cost. Moreover, this solution has the advantage to reduce the size of object identifiers, since they only need to name objects that reside in the same cluster (a lot less than the whole universe!). This naming scheme is used in the Mneme [Moss90], Thor [Day92] and Amadeus [Cahill93b] projects with 32 bit local identifiers.

  However, this scheme has a serious drawback: two different local identifiers in two different clusters can point to the same object, and the same local identifier in two different clusters can point to different objects. Then, at execution time, the system needs to associate a global name to each object identifier; an operation on an object identifier (assignment, comparison) requires a binding, if not to the object at least to a proxy, i.e a binding from the local identifier to a global one, even if the object is not used for method invocation. In the Amadeus system, when an object's data is made accessible in an address space, local identifiers are eagerly swizzled (i.e. replaced

by virtual addresses which are unique identifiers in the current address space); all the identifiers that may be used in memory need to be translated.

3.4 Object binding

3.4.1 Design

The main motivations in the design of our generic virtual machine [Freyssinet91] are to provide dynamic binding of references (in order to accommodate polymorphism rules of languages), and to support persistent shared objects that may be used to build more complex structures by embedding references to external objects within the instance data of an object. This current design is based on the following decisions:

- In the first prototype [Balter91], each method call was interpreted, i.e. the binding of code and data was checked by the kernel before the actual call. In order to improve performance, interpretation is now only done at first call.

- Since we only have a 32 bit address-space, we reuse space by dynamically mapping clusters in address spaces. An object may be mapped at different addresses, thereby precluding the use of traditional pointer swizzling. The solution was to simulate a Multics-like segmentation mechanism [Organick72].

A reference in an object *O1* to another object *O2* mapped in the same context *C* is made through a linkage segment associated with *O1* in this context. This linkage segment is built at the first use of *O1* in *C*, using a model generated by the compiler. For each external reference in *O1*, the compiler includes an entry in its linkage segment; this entry is filled (i.e. the reference is bound in *O1*) at the first method call from *O1* to the object pointed by this reference. After binding, further method calls to the object use indirect addressing through the linkage segment of *O1*, without further interpretation.

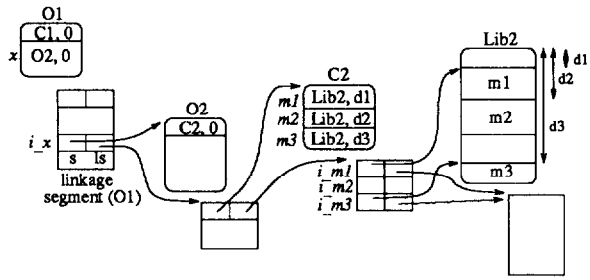

Figure 3: Object binding and method call

In fact, all the abstractions of the virtual machine are managed in this way. .A code-library refers to other code-libraries through its linkage segment, and a class-object refers to code-libraries in the same way.

In Fig. 3, object *O1* contains an external reference to object *O2* in a field *x* (or variable). When this reference is bound, the entry *i_x* associated with *x* in the linkage segment of *O1* points[3] to object *O2* in the current context. In the same way, class *C2* contains the external references to the code of the defined methods; these references are also dynamically bound. When all the involved references are bound, an object invocation from object *O1* to object *O2* is performed with pointer indirections through the linkage segment of *O1*, *O2* and *C2*. Thus, if *R* is a register that points to the linkage segment of the current object *O1*, then the invocation of method *m1* on the object pointed by *x* will execute the method at the address : **R[i_x].ls->ls[i_m1].s**

Some measurements were made with a set of applications written in the Guide language. An object call in the same protection domain when all the involved references are bound costs 4.4 μs[4], while an object fault costs between 22 and 55 μs according to the state of the caches managed in the kernel. This can be compared to the cost of a procedure

[3] An entry of a linkage segment has two fields *s* and *ls* that respectively point to the referenced segment and its linkage section.

[4] These measurements were made on Bull-Zenith P.C. 486 (33 MHz).

call on the same processor (0.9 μs) and to the cost of the virtual method call on a C++ object (1.5 μs) where sharing, persistence and protection are not managed.

We also measured the frequency of direct calls in the supported applications, and we obtained very good results (between 75 and 90 % for most of the applications). A more complete evaluation is given in [Chevalier93].

3.4.2 Lessons learned and related work

The segmentation scheme introduced in the second prototype significantly improved method call performance. A comparable approach has been used in the E project [Schuh90] where an object invocation can only be performed using a local variable (managed on the stack). This local variable is swizzled when first used as our linkage segments are.

This technique avoids the classical problem of pointer unswizzling when pointers are swizzled in the object's state and also allows the mapping of objects in several address spaces at different addresses. It may be compared to the solution adopted in the COOL v2 system [Lea93]. In this system, objects (in fact clusters) may be mapped in several contexts at the same time and all the objects identifiers in the cluster are eagerly swizzled. A cluster is always mapped at the same address in all the contexts that share it. If a memory clash occurs when a context tries to map the cluster, the cluster has to be unmapped in the contexts that map it, in order to allow its mapping at another address in the context that requested it. The main drawback of this solution is the complexity of unswizzling object identifiers when a cluster is unmapped.

Our approach can also be compared to the emerging one adopted in projects that aim to exploit the 64 bit address space processors. The key idea is to allocate at creation time the virtual memory area in which the object will be mapped when accessed. Therefore, an object is always accessed at the same virtual address irrespective of the process that shares the object, and there is no translation between object identifier and virtual addresses to do, since the object identifier is the virtual address where the object is or should be mapped. This new approach is very promising and many projects [Chase92][Heiser93][Inohara93] are in progress, but if this technique provides a fast addressing scheme, it has to be integrated in a real system that also takes care of other aspects such as protection and persistence. We plan to investigate 64 bit address space based systems in the future.

3.5 Object protection

3.5.1 Design

An important drawback in the first prototype was the lack of protection mechanisms for the development of protected applications. Mechanisms for the control of access rights on objects was an important goal in the second design. We placed the following requirements on the protection model:

- The system must allow the control of user's rights on shared objects managed in the system. These rights must be defined in terms of methods applicable to objects.

- The system must solve the delegation problem. In other words, it must be possible to extend a user's rights on an object for the execution of a specific operation.

  The game example given in [Kowalski90] illustrates this problem. An object *game* exports an operation *play*. Every user who wants to play invokes *play* on the object *game*. An object *score* is used to store the highest scores. The object *score* is updated using the operation *edit_score (user_id, new_score)* at the end of the game. Every user who has the right to play the game must have the right to call *edit_score* on object *score*, but a user must not be able to update *score* by invoking *edit_score* from an object other than *game*.

- The system must manage the cooperation between untrusted users. If user *U1* gives rights on his objects to user *U2*, *U2* must not be

given more than those rights. In particular, *U2* must not be allowed to tranfer these rights to another user. Additionally, *U1* must not get additional rights on *U2* (i.e. we don't want to manage a hierarchical organization of users since users are all equal regarding protection).

- Protection must be enforced without trusting the compilers used for application development.

Users' access control

In order to keep our efficient invocation scheme, we used the following design.

We defined the notion of *view* as a set of authorized methods. A view is a restriction of a class interface; the definition of views is stored in the class. The rights associated with an object are stored in an access control list (*Acl*) that defines for this object the view of itself that it provides to the existing users.

The general principle of our implementation is to use the protection information associated with an object to build an access path to an object at binding time, as seen in Multics. Thus the protection rights are set up when the object is bound, and remain valid for subsequent accesses.

At execution time, the access control according to such an access list is achieved as follows. For each class, a sub-section of its linkage segment is devoted to each view defined in the class. If the class defines NM methods and NV views, then the linkage segment of the class will contain NM*NV entries. We say that the linkage segment of the class contains NV views. In each of these views, the Nth entry corresponds to the same method, and the binding of the reference to this method in this view is only performed if the view definition in the class authorizes the method.

When the reference from an instance-object to its class is bound, the access list of the object is consulted to find out the view associated with the current user. Then the binding of this reference updates the linkage segment of the object[5] and makes

[5]The entry that corresponds to the class reference is always the first in the linkage segment of an instance.

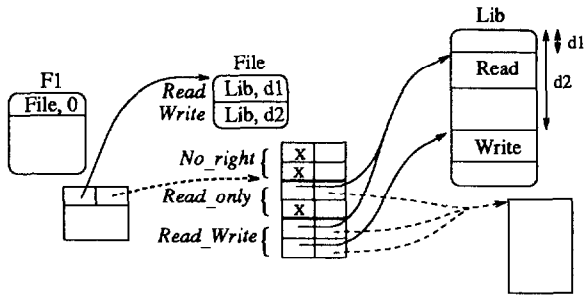

Figure 4: Protection based on access control lists

it point to the view associated with the current user.

This implementation is illustrated on Fig. 4. The class *File* defines three views: *No_right*, *Read_only* and *Read_Write*. An instance (*F1*) of the class *File* is mapped in a task that runs on the account of *User1*[6]. *F1*'s access list specifies that *User1* should use *F1* through the view *Read_Only*. Then, the binding of the reference from *F1* to its class points to the view Read_only in the linkage segment of the class *File*. In this view, an attempt to bind the second method will return an error.

With this implementation, the scheme used for an object invocation is unchanged. Protection checks are only made at binding time. However, a modification in an access list will only be taken into account in the next binding, but we think this is an acceptable trade-off between functionality and performance.

One of the requirements we made in the beginning of the section related to the safety of the provided mechanisms. In the Guide system, object isolation is provided through user isolation: objects owned by different owners are mapped in different contexts within a task. This implies that a method that executes on an object in one context will only have the possibility (if it breaks object encapsulation) to address objects that belong to the same

[6]Note that since contexts are never shared between Guide tasks, a context always runs on the account of a unique user and all the bindings in this context are made according to this user.

owner. Therefore, a user that runs a program can only corrupt its own data and can only access objects of other owners through methods, since access is achieved through an inter-context call. Note that the creation of an instance of a class implies trust by the user of the methods defined in that class.

Delegation problem

In the delegation problem, the purpose is to be able to extend user rights to some protected objects through some well defined entry points, the protected objects being not directly accessible. Since this ability is generally used to provide protected services, it has to be safe and this protection safety can only be obtained using context separation.

When an object invocation involves objects from different owners, it implies a cross-context call that is interpreted. Moreover, the owner of the calling object can be authenticated by the system, since an object owner is statically associated with each context in a task. The principle of our mechanism is to make rights depend on the calling object, and in particular to make rights depend on the owner of the calling object. A boolean tag called the visibility tag is attached to each object. This tag indicates whether the object to which it is attached can be invoked from an object owned by another owner.

In the example of the game, the game administrator creates the object *score* with a false visibility tag and the object *game* with a true visibility tag. The game administrator is the owner of *score* and *game*. So, when a player task invokes the method *play* on *game*, it executes *play* in a context associated with the game administrator, and object *score* can be invoked from object *game* because they reside in the same context. *Score* cannot be invoked directly from an object that belongs to the player.

The implementation of this mechanism consists in a simple check to verify, whenever an inter-context call involves different owners[7], if the called object has a true visibility tag.

[7] An inter-context call may involve two contexts associated with the same owner, but when these contexts are running on different nodes.

3.5.2 Lessons learned and related work

A more complete study of protection in object based systems in presented in [Hagimont93]. However, we now summarize the comparison with other approaches which justifies our design. Operating systems that provide mechanisms for access rights control may be roughly divided in two trends:

- Systems which make the rights depend on the calling object.

  A first instance of this class of systems is capability based systems (e.g. Hydra [Wulf74]). In capability-based systems, a method invocation is authorized if the calling object or method has a capability which authorizes the call. Access control for users and the extension of user's rights are easy to formulate, but capabilities have a drawback: when a server gives access rights to a client C, there is no way for the server to be sure that C will not give rights to some other clients on his own objects. When an invocation comes to the server from client C, the invocation may have been initiated by another client. Even if the server trusts C and if C is trustworthy, the server may not want to rely on the client safety, and the client may also not want the server protection to rely on the safety of its own objects.

  Another instance of this class of systems is systems which protection is based on access lists that contain object owners (e.g. Melampus [Luniewski91]). An access list is associated with each object and lists the users whose objects may invoke that instance. Then, they have similar problems to capability-based systems, since no check on the original issuer (user) of a request is made.

- Systems which make the rights depend on the calling user.

  In these systems, an access list is associated with each object and gives the users who may invoke that instance. Thus, an additional mechanism must be provided to solve

the delegation problem and allow the protection of subsystems. For instance, in Multics [Organick72], the ring's mechanism allows the development of protected subsystems, but not the management of mutually suspicious subsystems. If a subsystem *S1* is protected from a subsystem *S2*, *S1* must be in an ring inferior to *S2* and *S2* is not protected from *S1*.

In the Guide-2 system, we provide protection mechanisms based on access-lists that contain users, and the delegation problem is solved with the visibility-tag mechanisms. These mechanisms allow the development of mutually suspicious subsystems. The objects managed by such a subsystem belong to a pseudo-user (the administrator). Only the entry points of the subsystem (with a true visibility-tag) can be called by clients and access lists define the rights of the clients of the subsystem.

4 Conclusion and perspectives

In conclusion, we first summarize the lessons learned from our effort for providing shared object support in the Guide project. We next outline our plans and perspectives for the continuation of this work.

Summary

The basic message of this paper may be summarized as follows:

- Shared, transparently distributed objects, with high level language support, is an extremely useful tool for the development of distributed applications. We have implemented a generic system layer which allows the support of several object-oriented languages (Guide and C++).

- An object support system layer must provide a good trade-off between efficiency and protection. Fine-grained shared object support must not sacrify isolation between execution structures and between objects. In Guide-2, tasks are separate execution structures and object isolation is enforced on a per-owner basis. Moreover, a clustering scheme improves object management when logically related objects are grouped together.

- While the system must be able to manage a large amount of objects, the size of object identifiers must be kept small to reduce disk occupation and CPU time. Moreover, the system must provide migration facilities that allow objects to migrate between clusters. In Guide-2, objects are named by 64 bit identifiers; objects may migrate and the location time of a migrated object is shortened when its cluster is already mapped.

- An addressing scheme with lazy binding at first call allows the execution of most object invocations without any calls to the kernel. We simulated a Multics-like segmented machine which benefits from the locality of the references used for object invocations.

- In a system where objects may be shared between multiple users, it is indispensable to provide mechanisms for the control of user's rights on objects. Since our system does not rely on language provided safety, protection must be enforced by the system which must provide support for mutually suspicious subsystems. Protection in the Guide-2 system is based on access lists associated with objects and on visibility restrictions for the management of protected applications. The integration of these mechanisms does not incur any extra cost on method call within an address space.

Perspectives

From these two prototype implementations on Unix and Mach 3.0, we gained a more precise idea of what a micro-kernel should provide for the support of an object-oriented operating system such as Guide. A new project, alongside the Guide project, is under design and aims at providing this support. This kernel should be a common denominator for object support in distributed operating

systems and distributed data bases as well. It should integrate recent technology improvements such as 64 bit address spaces and high-speed networks which will allow a more intensive use of distributed shared memory with data replication and weak consistency.

Availability. Papers written in English describing the Guide system and the Guide language are accessible via ftp anonymous on the machine imag.fr. They are stored in the directory: /pub/GUIDE/doc

Acknowledgments. J. Cayuela, M. Riveill, and M. Santana contributed to the design and implementation of the Guide environment, including the work described in this paper. We also would like to thank S. Ritchie for reviewing this paper.

The work described in this paper has been partially supported by the Commission of European Communities through the Comandos ESPRIT project.

References

[Acetta86] M.J. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, M. Young, "Mach: a new kernel foundation for Unix Development", *USENIX 1986 Summer Conference,* pp. 93-112, July 1986.

[Bal87] H. E. Bal, A. S. Tanenbaum, *Orca: A Language for Distributed Object-Based Programming,* num. IR-140, Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, December 1987.

[Balter91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme, "Architecture and implementation of Guide, an object-oriented distributed system", *Computing Systems,* vol. 4, num. 1, pp. 31-67, Winter 1991.

[Balter93] R. Balter, P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte, X. Rousset de Pina, "Is the Micro-Kernel Technology well suited for the support of Object-Oriented Operating Systems: the Guide Experience", *2nd Symposium on Microkernels and Other Kernel Architectures (MOKA), San Diego,* September 1993.

[Black86] A.P. Black, N. Hutchinson, E. Jul, H. Levy, "Object structure in the Emerald system", *1st ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA),* Septembre 1986.

[Cahill93a] V. Cahill, R. Balter, X. Rousset de Pina, N. Harris, *The Comandos Distributed Application Platform,* Chapter 8, Springer-Verlag, 1993.

[Cahill93b] V. Cahill, S. Baker, C. Horn, G. Starovic, "The Amadeus GRT - Generic Runtime Support for Distributed Persistent Programming", *8st ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA),* pp. 144-161, October 1993.

[Chase92] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, Miche Baker-Harvey, "Lightweight Shared Objects in a 64-Bit Operating System", *7th ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA),* Octobre 1992.

[Chevalier93] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte, X. Rousset de Pina, "Experience with Shared Object Support in the Guide System", *4th Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS), San Diego,* September 1993.

[Dasgupta90] P. Dasgupta, R.C. Chen, S. Menon, M.P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R.J.

LeBlanc, W.F. Appelbe, J.M. Bernabeu-Auban, P.W. Hutto, M.Y.A. Khalidi, C.J. Wilkenloh, "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems,* vol. 3, num. 1, pp. 11-45, Winter 1990.

[Day92] Mark Day, Barbara Liskov, Umesh Maheshwari, Andrew C. Myers, *Naming and Locating Objects in Thor,* Laboratory of Computer Science, MIT, 1992.

[Decouchant91] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin, X. Rousset de Pina, "A synchronization mechanism for typed objects in a distributed system", *11th International Conference on Distributed Systems (ICDCS),* May 1991.

[Decouchant93] D. Decouchant, V. Quint, M. Riveill, I. Vatton, *Griffon: A Cooperative, Structured, Distributed Document Editor,* num. 93-01, Bull-IMAG, May 1993.

[Freyssinet91] A. Freyssinet, S. Krakowiak, S. Lacourte, "A Generic Object-Oriented Virtual Machine", *2nd International Workshop on Object Orientation in Operating Systems (IWOOOS), Palo Alto,* October 1991.

[Habert90] S. Habert, L. Mosseri, Vadim Abrossimov, "COOL: Kernel support for object-oriented environments", *5th ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA),* pp. 269-277, October 1990.

[Hagimont93] D. Hagimont, *Protection in the Guide object-oriented distributed system,* num. Accepted at ECOOP'94, Bull-IMAG/Systèmes, Grenoble, December 1993.

[Heiser93] G. Heiser, K. Elphinstone, S. Russel, G.R. Hellestrand, *A Distributed Single Address-Space Operating System Supporting Persistence,* num. 9302, School of Computer Science and Engineering, University of New South Wales, March 1993.

[Inohara93] S. Inohara, K. Uehara, H. Miyazawa, T. Masuda, *Sharing Persistent Data Structures on Wide Address Spaces in the Lucas Operating System,* Department of Information Science, Faculty of Science, University of Tokyo, July 1993.

[Jamrozik93] H. Jamrozik, *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'objets persistants,* Thèse de Doctorat en Informatique, Université Joseph Fourier (Grenoble), May 1993.

[Krakowiak90] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin, X. Rousset de Pina, "Design and implementation of an object-oriented strongly typed language for distributed applications", *Journal of Object-Oriented Programming (JOOP),* vol. 3, num. 3, pp. 11-22, October 1990.

[Kowalski90] Oliver C. Kowalski, Hermann Härtig, "Protection in the BirliX Operating System", *10th International Conference on Distributed Computing Systems (ICDCS),* pp. 160-166, May 1990.

[Lea93] Rodger Lea, Christian Jacquemot, Eric Pillevesse, "COOL: system support for distributed object-oriented programming", *Communications of the ACM, Special issue on Concurrent Object Oriented Programming,* vol. 36, num. 9, September 1993.

[Liskov85] B.H. Liskov, *The Argus language and system, Distributed systems: methods and tools for specification,* Lecture Notes in Computer Science, Vol. 190, Springer-Verlag, pp. 343-430, 1985.

[Luniewski91] Allen W. Luniewski, James W. Stamos, Luis-Felipe Cabrera, "A Design for

Fine-Grained Access Control in Melampus", *2nd International Workshop on Object-Orientation in Operating Systems (IWOOOS), Palo Alto,* October 1991.

[Moss90] J. Eliot B. Moss, "Design of the Mneme Persistent Object Store", *ACM Transactions on Information Systems,* vol. 8, num. 2, pp. 103-139, April 1990.

[Nguyen91] H. Nguyen Van, *Compilation et environnement d'exécution d'un langage à base d'objets,* Thèse de Doctorat en Informatique, Institut National Polytechnique (Grenoble), February 1991.

[Organick72] E.I. Organick, *The Multics system: an examination of its structure,* MIT Press, 1972.

[Schuh90] D.Schuh, M. Carey, D. Dewitt, "Persistence in E Revisited - Implementation Experiences", *4th International Workshop on Persistent Objects Systems,* pp. 345-359, September 1990.

[Wulf74] W.A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack, "Hydra: The Kernel of a Multiprocessor Operating System", *Communications of the ACM,* vol. 17, num. 6, June 1974.