Protection in the Guide object−oriented distributed system

Daniel Hagimont [(1)]

Bull−IMAG/Systèmes, 2 av. de Vignate, 38610 Gières − France

Internet: hagimont@imag.fr

Abstract:

Support for cooperative distributed applications is an important direction of computer systems research involving developments in operating systems, 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 shared by concurrent activities.

While many experimental distributed object−oriented systems have been designed and implemented (Gothic [Banâtre91], Clouds [Dasgupta90], Emerald [Hutchinson87], Argus [Liskov85]), few have addressed the development of protected applications in such an environment (Melampus [Luniewski91], Birlix [Kowalski90]).

In the Guide project, we have designed and implemented a distributed system that not only intends to support the above model, but also provides mechanisms for the development of protected cooperative applications. The advantages of the provided mechanisms are that they allow the cooperation of mutually suspicious users, that protection does not rely on the safety of the code produced by the compilers and that their implementation does not severely degrade object addressing performance. Protection in the Guide system is based on access lists and visibility restrictions on objects.

A prototype version of the system has been implemented using the Mach 3.0 micro−kernel as a base. Some simple applications have also been developed. More elaborate tools that allow the configuration of protected applications are to be implemented.

(1) The 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.

1 Introduction

Support for cooperative distributed applications is an important direction of computer systems research involving developments in operating systems, 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 shared by concurrent activities. In the Guide project, we have designed and implemented a system that supports such a model. Our goal is to provide an efficient platform for a family of object−oriented languages such as Guide (a language designed by our group [Krakowiak90]), and a persistent extension of C++. In particular, we wish to enhance sharing and protection, to simplify integration and to improve the performance of complex cooperating applications manipulating a large number of small objects (i.e. about a few hundred bytes). Our target application domain includes office applications, such as a cooperative document editor [Decouchant93] and a system for document circulation [Cahill93] composed of groups of interacting data centered tools that are inherently interdependent and have frequent interactions.

In a system where objects may be shared between multiple users that interact through a cooperative application, it is indispensable to provide mechanisms that allow the control of user's rights on objects. The second version of the Guide system (Guide−2) has been designed to provide such mechanisms; these mechanisms have been integrated into the objects' addressing scheme.

The lessons learned from our experience may be summarized as follows :

- We can provide access control mechanisms at system level without relying on the safety of the code produced by the compilers.
- We can implement access control on fine−grained objects without method call performance being severely degraded.
- Per object access lists that register users' rights (rather than capabilities) work well in the case of mutually suspicious users or applications.
- The delegation problem that arises with user based access lists may be solved with the definition of application entry points that allow the design of mutually suspicious sub−systems.

The remainder of the paper is organized as follows. Section 2 presents the requirements for access control in a system and the way existing systems fulfill these requirements. Section 3 describes our experience. It summarizes the main design choices and implementation principles of the Guide system, and then concentrates on the aspects related to protection. Section 4 presents a preliminary evaluation of these mechanisms. Section 5 is devoted to conclusions and perspectives.

2 Requirements and related work

The purpose of this section is first to present the most important requirements for the support of cooperative applications, and then to describe different approaches through previous system examples for providing this support.

The protection mechanisms we want to provide must deal with:

- Access control regarding users

  An application is always running on the account of one user. The system must allow the control of users' rights on shared data managed in the system.

- Delegation problem

  It must be possible to temporarily extend users' rights for the execution of a specific operation. The game example given in [Kowalski90] illustrates this problem. An object(2) *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*.

  A player should be able to update *score* invoking *edit_score* from the object *game,* but access to *score* from other objects should not be possible.

- Mutually suspicious users

  We want to manage 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).

In these requirements, we did not include the problems that relate to right revocation. It is a difficult problem that is not currently dealt in the Guide project, and that could be the subject of a separate paper by itself.

These requirements will now be used to analyse some previous systems.

In systems, protection rights can be represented as a matrix [Lampson71] where rows contain rights associated with users and columns contain rights associated with objects (Fig. 1).

(2) At this step, we don't make any supposition about the meanning of the words Object or Operation. An Object can either be a segment or a file.

	Obj1	Obj2	Obj3	......
User1		Op1 Op2		
User2			Op1	
User3		Op2		
......				

Fig. 1: Lampson Matrix

Two approaches are often used for implementation purposes. The first one consists in gathering protection information by column. An *access list* is then associated with each object and contains users' rights on this object. The second approach consists in gathering protection informations by row. A *capability list* is associated with each user and contains rights on objects for this user. We now study three variants of these basic solutions.

2.1 Hydra

Hydra *[Wulf74]* is an object−oriented capability−based system developed on a specific hardware at Carnegie−Mellon University. Some machine registers allow the use of capabilities through a restricted set of operations (load, store, copy, restrict_rights and call). A capability is composed of a unique identifier and a field that describes the authorized methods among the methods declared in the type of the referenced object. The state of an object is composed of a data part and a C−list which is a list of capabilities. A process that executes in an object can only use the capabilities stored in the C−list of that object and some capabilities received as parameters. As each method call switches the capability space (or address space) of the current process, we say that each object is managed in a different *protection domain*(3).

We now examine how the previous requirements are taken into account in the Hydra system :

- Access control regarding users

  A user application has the ability to copy capabilities, to restrict their associated rights, and to give them to other user applications. Each user owns an initial capability list which determines the objects the user may invoke. These objects may contain other capabilities that provide additional rights. But if a user cannot reach a capability on a given object, he will not be allowed to invoke that object.

- Delegation problem

  As each object in Hydra is managed as a distinct protection domain, an object (*game*) may contain a capability on another object (*score*) and possess rights on this object. No object except *game* will obtain the capability on object *score*. Without this capability, *score* cannot be accessed. The player will only get a

(3) *A protection domain may be viewed as an address space; method call is the only way for changing the execution protection domain.*

capability that allows the invocation of the method *play* on the object *game*. The invocation of method *play* extends the rights of the current process.

- Mutually suspicious users

  When a server gives access rights to a client, the server does not get rights on the client's objects. However, the client gets the right to give the capability to another client, which implies that the server looses some control on the rights it exports. The problem is not solved if clients are authenticated by capabilities, since clients may exchange these capabilities. This comes from the fact that no user check is performed at the system level.

Hardware−based capability systems have had limited use because they only run on specific hardware. Capability−based systems now running on classical hardware provide capability protection with encryption based algorithms *[Mullender86]*.

2.2 Multics

Multics *[Organick72]* is a system developed at Massachussetts Institute of Technology that also runs on a specific hardware. Protection in Multics is based on access lists. The basic unit of shared data is the segment, and an access list is associated with each segment and registers users rights (read/write/execute) on the segment.

Moreover, the system manages rings: there are 8 protection rings and a process always executes in one ring. A ring bracket is attached to each segment and gives for each operation (read/write/execute) the rings in which the operation may be performed. Brackets for read and write operations always start at ring 0, which means that a process executing in this ring has access rights for all the segments managed in the system. This scheme is in fact a generalization of the classical master/slave model, in which several protection levels are managed.

In order to allow processes to enter a lower ring, procedure segments may export some entry points (*gateways*) that may be called through a specific instruction. A process that calls a gateway from a procedure segment enters the higher ring of the execution bracket attached to the segment.

We now examine how the previous requirements are managed in the Multics system :

- Access control regarding users

  An access list is associated with each segment and controls users rights on the segment. Therefore, access control in terms of users is possible.

- Delegation problem

  In Multics, we can manage a data segment in a ring, i.e. the segment can only be accessed by a process that executes in this ring (the segment may contain the *game* and the *score* objects), and constrain processes that want to enter this ring to call a dedicated gateway in a procedure segment also managed in the same ring (this gateway is then the *play* operation). Then, the data segment is protected against direct access and can only be adressed by the procedure segment.

A protected application can therefore be managed in a lower ring, and a process that calls a gateway to enter this ring extends its rights for the time of the execution in the ring.

- Mutually suspicious users

  With access control lists associated with segments, rights that are received by a user cannot be given to another user.

  However, if a user wants to provide a protected service in a server, he needs to manage its segments in a ring inferior to the ring in which client processes execute. This means that the server will have rights on the segments managed by its clients (in some upper rings), and that mutually suspicious sub−systems cannot be implemented.

2.3 Melampus

Melampus *[Luniewski91]* is an object−oriented system developed at the IBM Almaden Research Center that provides access control mechanisms based on access lists, but it differs from others by the fact that its lists contain object owners. An object's access list gives the users whose objects may invoke that instance. The originality of their implementation is to take into account immediately a modification in an object's access list.

We also examine how the previous requirements are managed in the Melampus system. Melampus has similar problems to capability based systems since no check on the original issuer of a request is made. This allows an intermediate object to forward unauthorized requests to a server.

- Access control regarding users

  As in capability−based systems, access rights regarding users may be controlled. While a user cannot get rights on an object that is the target object or which is an object that has rights on the target object, this user will not be authorized to call the target object.

- Delegation problem

  The delegation problem can be solved by retaining the rights on protected objects for a privileged user (owner). In the game example, objects *game* and *score* belong to the game administrator. Object *game* accepts calls from objects that belong to other users, but object *score* only accepts invocations from objects that belong to the administrator of the game.

  A comparable mechanism is also used in the Birlix system [Kowalski90] in which another attribute (the class) of the calling object may be used in access lists. The access list of object *score* can then specify that only invocations from instances of class *Game* will be accepted.

- Mutually suspicious users

  The same problem arise as in capability−based systems. If a server gives rights to a client *C*, i.e the server accepts calls from objects that belong to *C*, there's 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 if *C* trusts others. Therefore, server's protection relies on client trust.

2.4 Conclusion

Protection rights may be given to users (*U*) associated with processes like in Multics, or to calling objects (*O*) like in Hydra or Melampus.

In both cases (*O* or *U*), the problem that arises is to be able to control rights according to the other invocation parameter (respectively the calling user and the calling object). If rights are given to users (*U*), the delegation problem needs to make rights depend on the calling object. If rights are given to object (*O*), mutually suspicious sub−systems implementation needs to control the calling user.

We believe that user control (*U*) should be used as the basis for the support of protected mutually suspicious cooperative applications. Therefore, access control in Guide is based on access lists that contain users' rights. An additional mechanism allows us to solve the delegation problem by specifying the entry points of each application.

The other conclusion of this section relates to protection safety. In the described systems, protection relied either on a specific hardware or on trusted compilers. As the Guide system aims at supporting untrusted compilers, we have to provide a minimal degree of isolation between processes and objects, and to authenticate parameters on which depend access rights.

3 Protection in the Guide system

The previous section has detailed the requirements and the related work, this section is devoted to the work achieved in the Guide project.

Section 3.1 summarizes the main design choices and implementation principles of the Guide system. It focuses on the object addressing scheme because the protection mechanisms has to fit well into this scheme. Section 3.2 concentrates on the mechanims that are provided and their integration in the Guide kernel.

3.1 Summary of the Guide design and implementation

A more complete description and justification of the Guide design is given in [Chevalier93c].

3.1.1 Object and execution model

The object model provided by the Guide virtual machine defines basic abstractions for building complex structures. The virtual machine [Freyssinet91] is intended to be used by the run−time system of object−oriented languages (in practice: Guide and an extended C++). The model defines three basic abstractions: instance−objects, class−objects, and code−libraries. The corresponding entities are potentially persistent; they are named by universal system references. Fig. 2 shows the organization of these entities.

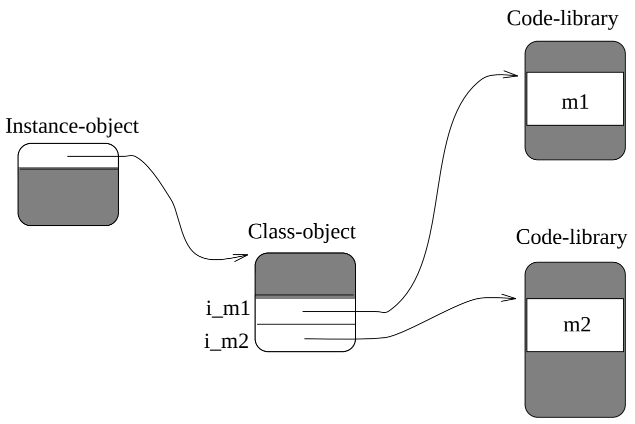

Fig. 2: The generic object model

Class−objects and instance−objects are defined separately, in order 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. The system does not manage relationships between class−objects (inheritance). The code of the methods involved in class definitions is stored in code−libraries.

Objects are named by unique system references, and may contain references to other objects. A code−library may contain a reference to a procedure in another code−library. Objects are passive (active agents are defined independently from objects).

The execution model is based on multi−threaded Tasks(4). A *Task* is a set of resources, in particular a distributed virtual address space, shared by its *activities* (sequential threads of control). The address space of a Task is composed of a set of contexts. A *context* is a virtual memory local to a node. A Task may span many nodes and the set of objects it contains may evolve dynamically. In practice, a program is represented by a Task, and a complex application may involve several cooperating Tasks.

Shared objects is the only means of communication between activities within the same Task or in different Tasks. The system should provide different policies to implement object sharing (i.e., one copy for read/write object−instances, multiple copies for class−objects).

(4) We use Task with a capital T to differentiate Guide Tasks from Mach tasks used in the implementation.

On top of the Mach3.0 micro−kernel, each context is implemented by a Mach task and a Guide Task is implemented by several Mach tasks distributed on the network. Mach threads are used to implement local representatives of activities in contexts and cross−context invocations are implemented using Mach IPC.

3.1.2 Management of shared objects

In order to be accessible, an object must be mapped in a context of a Guide Task. Object sharing between Tasks could be implemented either by sharing contexts between Tasks or by mapping an object in separate contexts, one per Task. The second solution was adopted in order to provide protection for individual objects. Tasks do not share contexts and protection is enforced by isolation of Tasks.

Furthermore, the system was also designed to provide object isolation: objects of different owners are mapped into different contexts in the same Task. When an activity spreads from an object owned by *X* to an object owned by *Y*, it must execute a cross−context invocation, which is interpreted by the system. Thus, an error in a method of an object can only affect objects having the same owner.

This design is illustrated on Fig. 3.

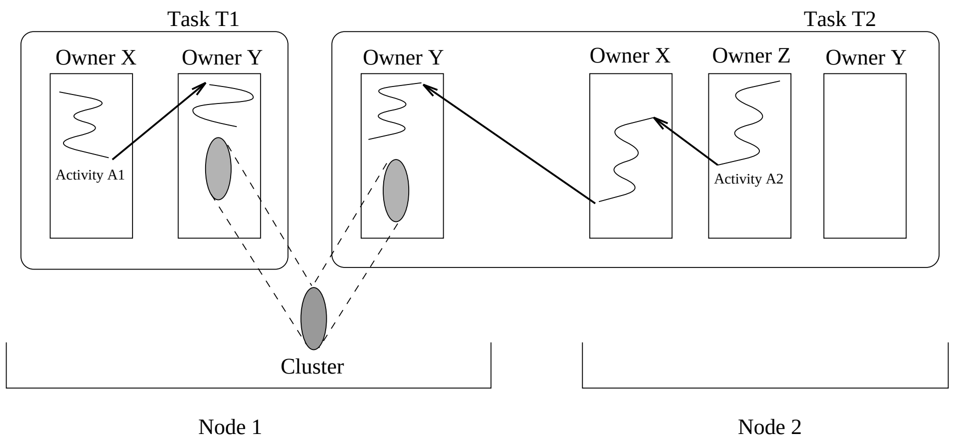

Fig. 3: Guide execution structures

Our experience shows that most Guide data objects are small (i.e. less than 300 bytes). Using 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 that have the same owner; clusters are the unit of mapping. A cluster is mapped in the context of a Task when two conditions are fulfilled: an object of the cluster has been called (for the first time) by an object mapped in the context; the object caller has the same owner as the called object. In practice, clusters can be used at the application level to group logically related objects; the cost of cluster mapping is amortized if most references are local to the cluster.

On Mach3.0, clusters mapping and sharing are implemented by the Mach external pager facility. A pager runs on each node and is in charge of the management of a set of clusters. These clusters are mapped in contexts according to the protection policy (isolation).

3.1.3 Object binding

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 design is based on the following decisions:

- In a previous 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.

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.

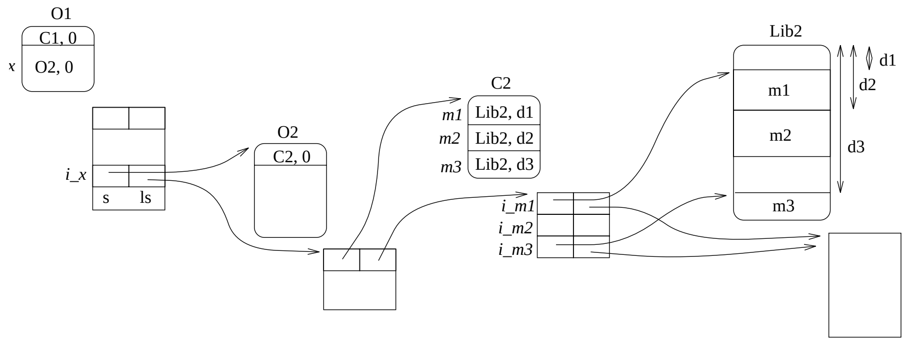

Fig. 4: Segmented mechanisms for object support

In Fig. 4, object *O1* contains in a field *x* (or variable) an external reference to object *O2*. As this reference has been bound, the entry *i_x* associated with *x* in the linkage segment of

O1 points(5) 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 *m2* on the object pointed by *x* will execute the method at the address :

```
R[i_x].ls->ls[i_m2].s
```

Note that after the variable *x* has been reassigned in context *C*, a context *C'* that shares object *O1* may have a linkage segment that still points to *O2* in *C'*. If this reference is used in *C'*, the reference should be explicitly rebound.

3.2 Design and implementation of the protection mechanisms

We must first summarize the requirements we had for the design of the protection mechanisms. These requirements are based on the study of protection in previous systems, but also on the design of the Guide system described above, since it was important for these mechanisms to fit well into this design.

As described in section 2, mechanisms based on access lists that contain users allow us to solve the problem of mutually suspicious users. Therefore, our approach was to provide access list based mechanisms and to solve the delegation problem with an additional mechanism.

The second requirement relates to the design of the Guide system, and especially to the addressing scheme we adopted. It was very important not to sacrifice the performance of object invocation for providing protection. In particular, we wanted to keep the binding at first call scheme.

Finally, the last requirement is the safety of the protection mechanisms. If it is acceptable that a user may bypass the protection system for his own objects, the system must guarantee that it is not possible for a user to corrupt or elude the protection system for objects from other owners. In particular in the context of the Guide project, in which a virtual machine is provided for the support of several object−oriented languages, the system cannot trust the code produced by the compilers and must enforce the protection mechanisms.

We now describe how the Guide system allows the control of access rights on objects using access control lists, and then the mechanisms used to solve the delegation problem.

3.2.1 Users access control

The fact that we cannot interpret each method call caused us to develop the following implementation.

(5) An entry of a linkage segment has two fields s and ls that respectively point to the referenced segment and its linkage section.

We define the notion of *view* as a set of authorized methods. A view is a restriction of a class interface which is stored in the class. For instance, in the class *File* that defines methods *read* and *write*, may be defined the following views:

```
Read_Write  :  (read,  write)
Read_only   :  (read)
No_right    :  ()
```

The access list of an object associates a view with each user. This is equivalent to the definition of the set of methods that the user may call on that object. An example of access list for an instance of the class *File* could be the following:

```
(  (User1,  Read_only),  (User2,  Read_Write),
  (Others,  No_right)  )
```

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(6) and makes it point to the view associated with the current user.

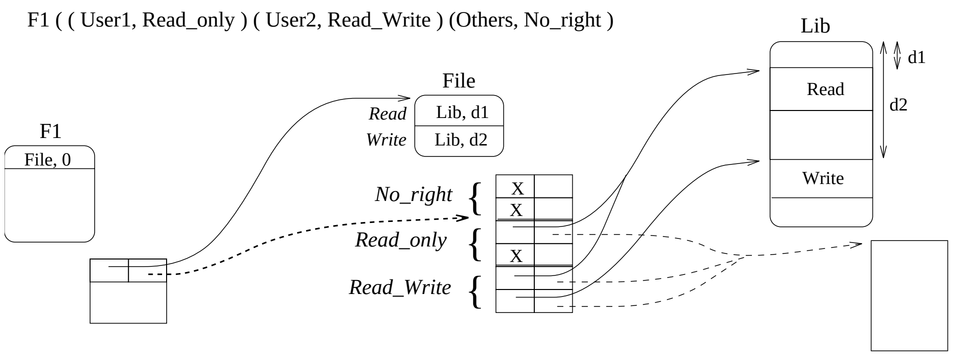

Fig. 5: Implementation of access control

(6) The entry that corresponds to the class reference is always the first in the linkage segment of an instance.

This implementation is illustrated on Fig. 5. An instance (*F1*) of the classe *File* is mapped in a context that runs on the account of *User1*(7). 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. *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. The binding of a reference from an instance to its class checks the rights stored in the access list of the instance. The binding of a reference from a class to a code segment (through one view of the class) checks the rights stored in the views definition in the class.

However, with this scheme, 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 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.

3.2.2 Delegation problem

As explained in section 3, a protection scheme based on access lists that contain users rights brings the need for an additional mechanism to solve the 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 protection domain separation (context separation in the Guide jargon).

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. The principle of our mechanism is to make rights depend on the calling object (as explained in section 2.4), 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

(7) The Guide Task (and each of its contexts) runs on the account of User1. The context may be associated with any owner (independently of User1).

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(8), if the called object has a true visibility tag.

We say that an object with a visibility tag set to true is an entry point of the application. This tag can also be set in a class; it defines the default tag for the instances of the class.

3.2.3 Object ownership

For the management of object ownership, it was possible to choose the owner of a created object as :

- the user associated with the executing Task,
- the owner of the object that requested the creation.

We chose the second solution because it is not well suited for mutually suspicious users to allow an object that belongs to user *X* to create an object that belongs to the caller *Y*. It would break the owner isolation we built, since it could create an object into which *Y* does not trust the implementation.

Moreover, it is more logical when complex structures are managed. If a text composed of chapters and sections is represented by an object tree (these objects belong to user *X*) and if user *Y* calls a method that adds a section in the text, it seems preferable that the section belongs to the owner of the text (*X*).

Therefore, an object is always created in the current context.

4 Evaluation

In this section, we try to provide a preliminary evaluation of the previous mechanisms, focussing on the following aspects:

- The adequacy of the defined mechanisms for programming protected applications. In particular, we detail their influence on the design of these applications.
- The impact of their implementation on the performance of the system.
- The safety of the protected applications.

(8) An inter−context call may involve two contexts associated with the same owner, but when these contexts are running on different nodes.

4.1 Adequacy

First, access lists are very convenient for managing protection rights according to users. They can be either managed at the language level or from a configuration tool.

Second, when an application is developed, the programmer has to specify the entry points of the application that can be reached from other applications. These entry points correspond to a consistent interface of the application according to its semantic.

We can then study two different ways to manage cooperation:

- If the application does not trust its users, then all the objects managed by the application will belong to the application manager. The protection is then very simple to program, using access lists and the visibility tag. Cooperating users can only enter the application through an entry point when its access list permit it. All the objects created by this application will belong to the application administrator, and will therefore be protected from illegal addressing.

  For instance, some system services such as user management may be managed in that way; objects belong to the system administrator, access lists authenticate the system managers and the visibility tag mechanism guarantees the consistency of the data.

- If the application manages objects that belong to its users, then the development of the application may be influenced by the protection mechanisms. An invocation inside that application where the callee and the caller objects belong to different owners will require a called object with a true visibility tag. This requirement has to be managed in the application, but it seems coherent to structure protected applications for inter−owner calls to go through some "consistent" entry points.

  In fact, in this case, the application may be viewed as a generic application; some instance of the generic application cooperates, each instance managing the objects of one user.

  An example of such an application could be a mailer application. The mailer has to manage objects that belong to different owners. If an object invocation occurs between two mailboxes that belong to different owners, then the application must be designed in such a way that a mailbox is an entry point of the mailer application.

  In fact, both of these approaches will be combined in application developments. Complex applications will manage objects that belong to the application administrator and also objects that belong to the application users. The application administrator is only a privileged user.

4.2 Performance

The table below gives performance figures for the different cases that may occur in the system. The machine is a Bull−Zenith P.C. 486 (33 MHz).

Object Call (without fault)	(1)	4.4 µs
Object Fault (object cached)	(2)	22 µs
Object Fault (object not cached)	(3)	55 µs

An object call without fault (1) does not call any primitive of the Guide kernel. This can be compared to the cost of a procedure 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.

In each context, a cache registers all the object bindings that occured. Then an object fault may find the object in the cache (2) or search the object in its location cluster (3).

More detailed measurements are given in [Chevalier93b].

The cost of protection is distributed as follows:

- Protection checks are done on object and method faults (so it does not put any overhead on direct object invocations).
- Linkage segments associated with classes are larger.

  This implies that the probability of finding the address of the called method in the linkage segment of the class is reduced.
- Invocations that involve objects from different owners cross context boundaries.

Some measurements on simple applications showed that most of object invocations are direct (without faults) [Chevalier93b]. We plan to also provide statistics about cross−context invocations.

4.3 Safety

As the system may support applications written with untrusted compilers, it has to assure the safety of the protection mechanisms.

The protection mechanisms provided in the Guide system allow a user to corrupt his own objects, simply by writing and executing an application that directly addresses virtual memory in the current context.

But mechanisms that intend to enforce user isolation and that protect against untrusted users are implemented with protection domains (Mach tasks). A user cannot directly address objects from other owners; he can only properly call a method on an object through an entry point.

In the implementation of the Guide kernel, the system has to authenticate users and contexts. It has to be sure that a user will not be able to give the illusion that he is somebody else or that a cross−context invocation comes from another context. This authentication of users and contexts is based on Mach protected port [Chevalier93a].

5 Conclusions and perspectives

In conclusion, we first summarize the basic design choices for providing protection mechanisms in the Guide object−oriented system. We next outline our plans and perspectives for the continuation of this work.

Basic design choices

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

- Protection mechanisms based on access lists are well suited for the support of cooperative applications developed on object oriented systems. Another mechanism is then used to solve the delegation problem.
- These mechanisms can be implemented at the operating system level:
    - Without trusting the supported compilers.
    - With a good trade−off between performance and safety.

The main design decisions that relates to protection are the following:

- Protection based on owner's isolation. Objects that belong to different owners are mapped in different contexts.
- An addressing scheme à la Multics. This scheme allows users to only pay the cost of an interpretation at first call; further invocations do not call any primitive of the Guide kernel.
- Views definition in classes. A view describes a set of authorized methods. A sub−table in the linkage segment of the class is associated with each view.
- Access lists associated to objects. An access list associates a view of the object's class with each user. The binding of the reference from an object to its class depends on the view attached to the current user.
- Access control is realized at binding time, when an object or method fault occurs.
- The delegation problem is solved with the visibility tag. This boolean tag is attached to each object and indicates whether the object can be called from an object that belongs to another owner.

The described mechanisms have been implemented in the Guide kernel and simple hand−coded protected programs have been written. However, we have not yet deeply explored the design of protected cooperative applications, since development tools that use these mechanisms are to be implemented.

Perspectives

The system provides a generic virtual machine for the support of several object−oriented languages. The applications we developed were written with the Guide language. We are currently working on the support of an extension of the

C++ language that would manage shared persistent objects. We are also working on the improvements to the Guide language.

We plan to develop tools that help in the development of protected applications. Some of the protection mechanisms could also be manipulated through an instruction set in the Guide language.

Next, we will be able to experiment through larger protected cooperative applications for the validation of our concepts.

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, P.−Y. Chevalier, A. Freyssinet, S. Krakowiak, S. Lacourte, M. Riveill, X. Rousset de Pina and M. Santana contributed to the design and implementation of the Guide environment, including the work described in this paper. Especially, I would like to thank X. Rousset de Pina and J. Mossiere for numerous discussions and helpful comments. I also would like to thank D. Acton 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.

Bibliography

[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 and G. Vandôme, Architecture and implementation of Guide, an object−oriented distributed system, *Computing Systems*, 4(1), pp. 31–67, Winter 1991.

[Banâtre91] J.P. Banâtre and M. Banâtre, *Les systèmes distribués, Concepts et expérience de Gothic*, InterEditions, 1991.

[Cahill93] V. Cahill, R. Balter, X. Rousset de Pina and N. Harris, *The Comandos Distributed Application Platform*, Chapter 8, Springer−Verlag, 1993.

[Chevalier93a] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Lacourte and 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.

[Chevalier93b] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte and 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.

[Chevalier93c] P.Y. Chevalier, A. Freyssinet, D. Hagimont, S. Krakowiak, S. Lacourte and X. Rousset de Pina, *Supporting shared persistent objects in a distributed system*, Technical Report, Bull−IMAG, Grenoble, May 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 and C.J. Wilkenloh, The Design and Implementation of the Clouds Distributed Operating System, *Computing Systems*, 3(1), pp. 11–45, Winter 1990.

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

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

[Hutchinson87] Norman C. Hutchinson, *Emerald: An Object−Based Language for Distributed Programming*, PhD thesis, University of Washington, January 1987.

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

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

[Lampson71] B.W. Lampson, Protection, *Fifth Annual Princeton Conference on Information Sciences and Systems*, pp. 437−443, March 1971.

[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 and 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.

[Mullender86] S.J. Mullender and A.S. Tananbaum, The design of a capability−based distributed operating system, *Computer Journal*, 29(8), pp. 289−299, August 1986.

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

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