Protection in an object-oriented distributed virtual machine

D. Hagimont, S. Krakowiak, X. Rousset de Pina

Bull-IMAG, 2 av. de Vignate, 38610 Gières, France

{hagimont, krakowiak, rousset}@imag.fr

Abstract. This position paper describes the principle of a protection model based on objects and its implementation on an object-oriented distributed virtual machine. The model is based on access lists, method groups, and visibility restrictions. It is currently being implemented on the Guide-2 system.

1. INTRODUCTION

This paper describes a protection model for an object-oriented system and its integration in a system currently being implemented to support object-oriented distributed applications.

In a broad sense, protection refers to the means by which the integrity and security of information is achieved. A general design rule that applies to protection is the separation between mechanisms and policies. Our goal is to define a basic protection mechanism to be integrated in an object-oriented system. More specifically, we are currently engaged in the design and implementation of a distributed object-oriented virtual machine, i.e. a system interface based on a generic object model, to be provided on a set of interconnected computers. The aim of this paper is to discuss the design and implementation of a protection mechanism to be integrated in this virtual machine. The definition of the protection policies that will be enforced by means of this mechanism is outside the scope of this paper.

More precisely, we aim to define a protection model with the following requirements:

- The units of protection should be individual objects; the model should support users and groups.
- The protection model should be consistent with the object model; in other words, access rules should be defined in terms of the access methods applicable to objects rather than in terms of read, write or execute actions.
- The model should provide isolation between users (i.e. an error in an user's object should only affect this user's objects) and between applications (an error in an application should not affect applications that do not share objects with it).
- The model should solve the delegation problem. In other words, it should be possible to extend a user's rights on an object for the execution of a specific operation. The game example given in [Kowalski 90] illustrates this problem. A class *game* exports a method *play*. Every user who wants to play invokes *play* on an instance of *game*. An object *score* is used to store the players' highest score. The object *score* is updated using the method *edit_score(user_id, new_score)*. A player should be able to update *score* invoking *edit_score* from its instance of *game*, but access to *score* from an object of an other class should not be possible.

  A general solution to the delegation problem would be to specify access permissions in terms of the class of the calling object, and not only in terms of the called object.
- The protection model should fit in the framework of the existing design for the object virtual machine.
- The model should be amenable to an efficient implementation.

The rest of the paper is organized as follows. Section 2 is a brief overview of the Guide object-oriented virtual machine. Section 3 presents the protection model. Section 4 describes the implementation of the protection mechanism.

2. OVERVIEW OF THE GUIDE VIRTUAL MACHINE

In this section, we give a brief overview of the Guide object and execution model, and of the object virtual machine currently being implemented to support this model.

2.1. Object and execution model

The object model is embodied in the Guide language ([Krakowiak 90]). The language is strongly typed and has separately defined types and classes. A *type* defines an interface as a set of attributes (simple variables) and method signatures. A *class* is an implementation of a type; it defines the program of the methods. Objects are generated as instances of classes.

Types may have subtypes, and classes may have subclasses; the type and class hierarchies are defined in parallel, with single inheritance. Types are statically checked at assignment and parameter passing, using conformity rules; in some cases, a dynamic check is necessary.

Objects are passive. The execution unit is a *job* (this roughly corresponds to an "application"). A job is composed of a (distributed) virtual space, in which one or several *activities* (sequential threads of control) are executed; activities are explicitly created according to a **cobegin-coend** scheme. Objects are dynamically bound into a job's virtual space as a result of method calls; jobs and activities spread out to remote locations if they need to access remotely located objects. The virtual space of a job is composed of several *contexts* (virtual address spaces), possibly distributed on several machines; this distribution is transparent. The location of the objects is determined by the system according to a location policy, currently fixed by default.

Communication between jobs and activities is by means of shared objects; there is no explicit message passing. Objects are persistent (i.e. an object's lifetime is not related to that of the jobs or activities which use it). In order to start an application, a user needs to specify an initial object and an initial method of this object. A job is then created, the initial object is linked within this job, and an activity is started with a call on the initial method. Other objects are then linked into the job as needed according to the calling pattern. The application may explicitly create new objects, new activities, and new jobs.

2.2. The virtual machine

A first implementation of the above model ([Balter 91]) was done on top of Unix and was closely tied to the Guide language. The object virtual machine has been redesigned ([Freyssinet 91]). The new version should provide generic support for object-oriented languages that satisfy the following set of minimal assumptions (as seen from the operating system):

- The language is class-based; a class describes the methods applicable to its instances and the internal organization of the instances; there is an explicit link from an instance to its class.
- Classes are organized in a hierarchy by the *is-a-subclass-of* relationship; there is no assumption about the organization of the inheritance graph (e.g. single or multiple, etc).
- Objects are named by universal references (i.e. references which are independent from any addressing context); this allows persistent objects to be defined.

The system has been designed as a hierarchy of abstract machines; for the purpose of this presentation, we are only concerned with two of those: the *segment machine*, which provides a segmented persistent memory; the *object machine*, which uses segments to build objects.

The segment machine uses indirect method calls in order to allow dynamic binding of the code of the methods, as required by objects models. The mechanism must support persistence, sharing, and distribution. Therefore, we must uniformly resolve references from different segments to a shared segment, which may simultaneously be mapped in several contexts, possibly on different machines, and at different addresses.

The proposed solution relies on a linkage section (à la Multics). This section is created each time a segment is mapped in a context, and it contains all context-dependent, non-persistent informations associated to the segment, including its mapping address. When the segment is unmapped or deleted, its linkage section is deleted. The main cost is incurred at binding time, when the segment is mapped in a context. This occurs at the time of the first call on a method of the object in the current context; subsequent accesses to the mapped segment are not interpreted, and use indirection through the linkage section. The additional cost is one level of indirection.

This scheme has the additional advantage that the internal representation of a persistent segment does not depend on whether it is mapped or not, thus avoiding the "pointer swizzling" costs incurred if references to other segments must be changed. References are universal (i.e. system wide) internal identifiers. They are interpreted by the system at binding time (i.e. at first call). The system locates the segment using its reference; the segment is loaded from the storage system if not mapped in the current context. The segment may already be mapped in another context, in which case the execution policy determines whether it should be remotely called or remapped. Non-persistent segments, which are only used for communication, may also be defined.

In order to reduce the cost of mapping, segments are grouped into clusters, which are the units of mapping. The criteria for clustering are not discussed in this paper.

3. THE PROTECTION MODEL

3.1. Definition of the main concepts

- *User*. A user is an administrative entity. It is named by the system using an identifier (*Uid*) that is unique in space and time.
- *Owner*. Every object has an owner defined by an owner-tag whose value is the owner *Uid*. The owner-tag of an object is equal to the owner-tag of the object that has created it. The Guide virtual machine provides a primitive that allows system administrators to change the owner-tag of an object (equivalent to the *chown* of Unix).
- *Group*. A group is a set of users that is named by an *Uid*. Each user can be seen as a group reduced to itself; a user may be a member of several groups.
- *View*. A view is a subset of the methods that can be invoked on the instances of a class. A view description is attached to a class object. A class object exports an arbitrary number of views on its instances.

3.2. The protection model

Each job is operating on the account of a user called its owner and on the account of a group to which the owner belongs. The group associated with a job is (by default) the group inherited from its job creator or another group requested at creation time. A system process, the *group manager*, controls group allocation to jobs. It must authenticate the owner's *Uid* and verify that the owner is a

member of the requested group. The group manager has a key position in our protection schema, because the access rights of a job depends on its group.

Jobs do not share contexts, which guarantees mutual isolation between jobs running at the same time and on the same node. Objects of different owners are mapped in different contexts in order to enforce isolation. 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. Compared to a solution in which all method calls would be interpreted, this seems to be an acceptable trade-off between security and performance.

The rights associated with an object are implemented by an access list (*Acl*) that defines for each object the view of itself that it provides to the existing groups. An object's *Acl* can only be modified by an invocation from an object having the same owner. Modifications of the rights of a group on an object have no effects on group members that have already mapped the object, but will be seen by every job that subsequently attempts to map the object. If the owner of an object has not specified an *Acl*, the object can only be accessed by its owner and without view control (i.e. the owner can invoke all the methods exported by the object's class). An object can be declared unprotected, which means that all the existing groups have the same view of the object.

Such a definition of *Acl* entry in terms of views provided to a user on an object does not allow to solve the delegation problem. To solve this problem, the Birlix system [Kowalski 90] makes the righs of a group on an object dependent on the class of the invoking object. Thus, when an object is accesssed, not only the group's *Uid* but also the invoking object's *Uid* are checked to know whether the invocation can proceed. As we show in the next section, our implementation of the invocation mechanism does not allow us to trust the *Uid* of the calling object, so we cannot apply this model. In our model, a boolean tag, called 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 an other owner that its owner.

The isolation between objects of different owners together with the visibility tag allows to solve the delegation problem. In the example of the game, the game administrator creates the object *score* with a false visibility tag and an instance of the class game per player with a true visibility tag. The game administrator is the owner of *score* and the instances of the class *game*. So, when a player job invokes the method *play* on an instance of *game*, it executes *play* in a context of its job associated with the object of the game administrator. A player cannot forge his score because he cannot access the object *score*. This solution is less accurate and less flexible than the solution proposed by Birlix and can be compared to the Unix *setuid* because it implies the creation of one pseudo owner by application. However it fits very well in our invocation schema, as is shown in the following section.

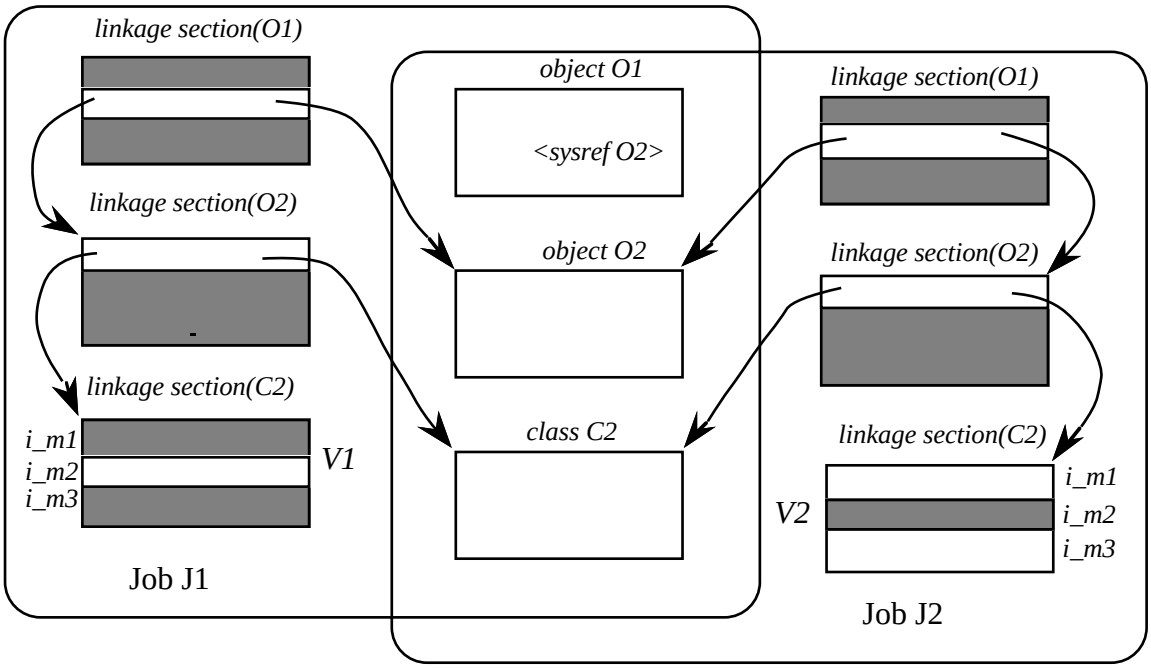
4. IMPLEMENTATION OF THE MODEL

The general principle of our implementation is to use, like in Multics ([Organick 72]), the protection information associated with an object to build an access path to an object when the object is bound in a context. Thus the protection rights are set up when the object is bound in a context and remain valid for subsequent accesses.

The following figure illustrates how an access path is built from the object *O1* to the object *O2*, owned by the same owner *P*, in the jobs *J1* and *J2* sharing these objects. The jobs *J1* et *J2* are respectively running on the account of user *U1* (resp. *U2*) and group *G1* (resp. *G2*).The *Acl* of *O2* gives *G1*'s members the view *V1* (*m2* is the only method that *G1*'s members may invoke) and *G2*'s members the view *V2* (*m1* and *m3* are the methods that *G2*'s members may invoke).

The information that define an object's protection is itself mapped in the same context as the object and is not protected. This can only affect the objects of the same user, since calls to other users' objects must cross a context boundary. However, another consequence is that the system cannot trust the informations about the class of a calling object, which explains the visibility tag approach.

The views' description are stored in the object class and are used to build the appropriate linkage section when the object class is mapped in a context. The access list associated with an object is implemented in the same cluster as the object. Since a cluster is the mapping unit in a context, all the objects included in a cluster must have the same owner.



5. CONCLUSION

We have presented the design and implementation principles of a protection mechanism to be integrated in an object-oriented distributed virtual machine. Let us compare this approach with recent work on protection.

Protection in Amoeba [Tanenbaum 90] is based on *capabilities*, which provide a uniform mechanism for naming, accessing and protecting objects. Amoeba capabilities include a 8-bit field which defines the set of operations that its owner can call on the named object. The capabilities which refer to an object can only be modified by the owner of the object; they are protected by a cryptographic method. Our model differs from Amoeba on two main points: objects in Amoeba are not explicitly based on classes; Amoeba is based on a client-server model, not on distributed shared memory. Using capabilities in Guide would mean interpreting every call to an object.

Protection in Birlix is based on access control lists (*Acl*s) and subject restriction lists (*Srl*s). In order to solve the delegation problem, *Acl*s may have entries for types as well as for users. *Srl*s are used to speed up the introduction of new users, by avoiding to scan all the existing *Acl*s. In addition, Birlix provides a mechanism that allows to determine the responsibility for any action. Our solution to the delegation problem is close to that of Birlix; however, our granularity of control is limited by efficiency constraints, since we do not want to interpret each method call.

The model is currently being implemented as part of the Guide-2 system, based on the Mach 3.0 micro-kernel. Experience with the prototype will allow us to check our claim of achieving an acceptable tradeoff between safety and efficiency.

Acknowledgments.

Philippe Ingels (IRISA, Rennes) was a major contributor to the definition of the protection model. Jacques Cayuela, Pierre-Yves Chevalier, Andrzej Duda, André Freyssinet, Serge Lacourte, Michel Riveill and Miguel Santana have contributed to the design of the Guide object support system. This work was partly supported by the Commission of European Communities under the Comandos ESPRIT project (nr 2071).

REFERENCES

[Balter 91]
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, 1, pp. 31-68

[Krakowiak 90]
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*, 3,3, (Sept.- Oct. 1990)

[Kowalski 90]
O. C. Kowalski, H. Härtig, Protection in the Birlix operating system, *Proc. Int. Conf. on Distributed Computing Systems*, May 1990, pp. 160-166

[Organick 72]
E.I. Organick, The Multics system: an examination of its structure, *MIT Press*, 1972

[Freyssinet 91]
A. Freyssinet, S. Krakowiak, S. Lacourte, A generic object-oriented virtual machine, *Proc. Int. Workshop on Object-Orientation in Operating Systems*, Palo Alto, October 1991, pp. 73-77

[Tanenbaum 1990]
A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, G. van Rossum, Experiences with the Amoeba distributed operating system, *Comm. of the ACM*, 33, 12, December 1990