Introducing Distribution in an OS Environment of Reflective OO Abstract Machines

Fernando Álvarez García, Lourdes Tajes Martínez,
María Ángeles Díaz Fondón, Darío Álvarez Gutiérrez
Departamento de Informática, University of Oviedo
falvarez@correo.uniovi.es

Abstract

Distribution is a key property for future Integral Object-Oriented Systems (IOOS) based solely on the Object-Oriented paradigm. The introduction of object distribution in the IOOS is not trivial if we want a clear distinction between mechanisms and policies and even more if we want policies to be easily replaced or customized. We have found that reflection is a well-suited technology for endowing the IOOS with such a distribution property.

1 Introduction

The adoption of the object-oriented paradigm is growing steadly. With the Internet boom and distributed systems, a heterogeneous distributed interoperable object environment is appearing. Platforms and architectures such as Java and CORBA are responsible for the evolution towards this environment, increasingly seen as an environment for the future.

An integral object-oriented system (IOOS) is an example of this kind of "world of objects" environments, based solely on the OO paradigm, which are hinted as the evolution of current environments. An IOOS ideally provides a virtually infinite space where objects live indefinitely and exchange messages regardless of their location.

The Integral Object-Oriented System Oviedo3 is an IOOS under development at the University of Oviedo. Oviedo3 is structured upon an Object-Oriented Abstract Machine (OOAM) and an Object-Oriented Operating System (OOOS) [ATA+97]. The OS objects transparently extend the basic object model provided by the machine.

Abstract machines provide basic support for objects, allowing object creation, invocation, deletion, etc., but do not know the existence of other abstract machines in the network. We want to keep abstract machines as simple as possible but with the possibility of extending them transparently with properties like persistence or **distribution**.

Distribution in object-oriented systems is basically defined by two terms: object mobility and remote invocation support. Where to put the distribution support is an important design issue. Some systems leave it at the programming language level [Jul88]; some others, at the operating system level [DLA+91]. It is also possible to implement distribution support as a middleware technology [OMG99]. Distribution policies are usually mixed with mechanisms, and with no provision for their customization.

The idea explored in this paper is the use of *reflection* to endow an IOOS with the distribution property. Distribution will be an operating system issue, and the underlying abstract machines will not be aware of it. Abstract machines will ask for help to operating system objects in a reflective way to solve every distribution-related problem they cannot deal with. The implementation of distribution

mechanisms and policies at the operating system level make them replaceable, customizable and extensible, taking advantage of the OO paradigm built in the system.

The rest of the paper is organized as follows. First, the basic architecture of the IOOS is presented in section 2. Next section describes how reflection is introduced in the abstract machine. Section 4 presents some initial distribution issues. Sections 5 and 6 describes how remote method invocation and object migration can be achieved and section 7 proposes solutions for some advanced issues. Section 8 describes the current implementation status. Conclusions are drawn in the last section.

2 The abstract machine architecture

The machine supports an object model with the following features: object identity and abstraction, encapsulation, inheritance, and also generic and aggregation relationships between objects, message passing with polymorphism. Concerning object to process relationship, an active object model, where several threads can exist simultaneously in every object has been adopted to simplify object persistence and migration. References are the only means to access objects, and include the identity and class of the object pointed to. Computation is achieved by message passing between objects.

Figure 1 shows the logical architecture of the abstract machine, named Carbayonia, consisting of four areas: classes area, references area, instances area and system references area.

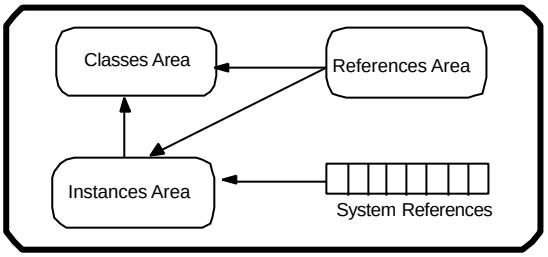

Figure 1. Architecture of the Carbayonia abstract machine.

Every object is represented using the structure shown below:

Object identifier
A reference to its super-object (if it is a subinstance)
A list of references to its aggregated objects
A list of references to its associated objects
A list of references to its meta-objects
A reference to its class
A reference to its shadow class

References to meta-objects are needed for the reflection property of the machine. The *shadow class* is an on-demand created class for the object. It will store copies of object attributes and/or methods (of its class or inherited) that are not locally accessible or for performance reasons.

3 Reflection in the abstract machine

The reflective architecture of the abstract machine [TAD+98] divides the space of objects in two levels: the *base level* and the *meta-level*. Objects at the base level implement user applications. The meta-level is decomposed in a set of objects called meta-objects, thus maintaining the imposed OO paradigm. Objects at the meta-level extend, modify or complement the default behaviour of the machine. Operating system issues are basically implemented at the meta-level.

Reflection is introduced in two ways: structural reflection and behavioural reflection. Structural reflection is focused on exposing, as normal objects, the inner architecture of the machine and object

implementation at run-time. This way, user objects can invoke instances- area methods, classes-area methods, etc. They can obtain, as well, their composition: class, superclasses, attributes, methods, etc.

Behavioural reflection is focused on exposing the behaviour of the machine. Machine components responsible of base-level object execution are reified by means of a set of meta-objects: the *messenger* meta-object for object invocation, the *synchronizer* meta-object for inner concurrency management and the *scheduler* meta-object, responsible for the scheduling of a set of active threads.

Regarding distribution, the messenger meta-object is the most important one. It is actually a pair of meta-objects: a *sender* and a *receiver*. The sender meta-object controls every issue related to message dispatch. It makes possible to customize the set of actions to carry out before and after sending the message. The receiver meta-object controls every issue related to message reception by an object. The receiver meta-object can make decisions such as delaying method execution, delegating or rejecting the message.

4 Distribution design issues

Interactions between objects are based solely in object invocations. A location independent, global object identifier uniquely identifies each object. An object location mechanism must be provided for locating objects with the object identifier as the unique information.

Distribution transparency is important for simplifying most of the applications. However, there are applications that need some (even full) control over distribution, so facilities for distribution control must also be provided.

The self-containment property of active objects facilitates object migration. When moving an object, not only its state but its computation are moved. This raises the question of whether other objects should also be moved or not. Objects store relationship references to other objects that will help to answer this question.

How to pass parameters in object invocations is another important design issue. Parameters will be passed "by reference", the natural method. If the invocation is remote, the use of the parameter objects will be remote as well. To avoid such additional remote invocations, an optimization can be made at the meta-level to achieve a call-by-move parameter passing semantics [Jul88].

Finally, all these issues must be compatible with the object model supported by the abstract machine: encapsulation, inheritance, etc. must be maintained regardless of the location of objects, classes and references.

5 Remote method invocation

Remote method invocation is basically supported by the behavioural reflection property. To sum up, the sender and receiver meta-objects will be modified to extend the local method invocation mechanism provided by the abstract machines to a global method invocation one.

The sender meta-object is given the control by the abstract machine each time an object invocation takes place. If the invoked object exists locally (a query to the instances area reflected object suffices), the receiver meta-object of the invoked object is informed to carry out the server part of the invocation. The receiver meta-object must ensure that a local description of the method being invoked exists (a description of the method could be unavailable if the object had been moved far from its class or any of its superclasses definition). If the description of the method is not available, a *method fault* is generated and managed, so that the method is located in the integral

system and retrieved. The copy of the method is placed in the shadow class of the object. Finally, the receiver meta-object asks the synchronizer meta-object for creating a thread for the invocation.

If the invoked object does not exist locally, it must be located. An operating system object, called *locator*, is invoked to obtain the location information needed to forward the request. The locator object executes a strong location policy that fixes and possibly activates (if the object is in persistent storage) the desired object.

A message containing the invocation information is created with:

- ? a reference to the invoking object,
- ? the copy of the reference to the invoked object owned by the invoking object,
- ? a list of references that represents the list of invocation parameters,

and sent to the destination machine by means of a *communicator* object that deals with the network particularities.

A peer communicator object in the destination machine receives the message and creates a *worker* object, which converts a remote method invocation into a local one. We are now in the basic case, where invoking and invoked objects reside in the same machine.

6 Object migration

When talking about object migration, some questions must be answered: when, how much, where and how to move. The first three are policy issues. The last one is the mechanism itself.

If objects have not distribution control, they will be moved only because of operating system decisions (load balancing, reduction of network messages, etc.). If we want objects to have some distribution control, we must give them a means of requesting movement operations.

The unit of migration is the object. But we must also decide what to do with its related objects. It seems reasonable that moving an object implies moving its related objects, too. However, the cost of the whole movement must be taken into account. Again, the actual policy will decide.

Where to move is very related to when to move. Usually, whatever the reason the operating system has to move an object, it also decides the destination node of the movement. For example, if two objects have a strong interaction, one of them will be moved to the other's location to minimize network usage.

Finally, how to move means describing the set of operations that result in the transparent disappearence of an object in the origin machine and its appearence in the destination machine. Transparency here means that the object will continue its computation at the destination node as if no movement had been taken place and that invocation requests made when the object was moving will be redirected to its new location.

An operating system object, called *mover*, will be invoked any time an object must be moved. The mover object will execute the next set of operations to complete the movement:

1. Obtaining a copy of the encapsulated state of the object. To obtain such a copy, the moving object must be taken to a consistent state, where no threads are in the running state and arriving invocation requests are blocked. The mover object asks the synchronizer meta-object to proceed this way. When the object is in a consistent state, the mover asks the instances area reflected object for a string containing it (serializing the object's state and computation). The string includes the object identifier, all the atributes references, the reference to its class,

its shadow class (with the copy of every method with executing threads) and references or copies of its meta-objects (depending on whether they are shared or private).

2. The string copy containing the object state is sent to the destination machine by means of the local communicator object. A remote communicator object will receive the message at the destination machine and gives it to a worker object. The worker object will be responsible for completing the migration operation.
3. The worker object asks the instances area reflected object to create an object from the string. The object will be created with the identifier contained in the string (an exception in the usual identifier assignment) and put into a suspended state, provided that the original copy has not been deleted yet.
4. The movement completes with the deletion of the original copy of the object, the object location information update (if needed) and its reactivation. The reactivation process consists of invoking the synchronizer meta-object to unblock every pending invocation requests and to resume every thread that was in a running state.

As it could be seen, object migration is supported by both the structural and behavioural reflection properties of the machine. The abstract machine and the object been moved are unaware of the migration operation, that takes place at the meta (operating system) level. Objects that need some distribution control have, firstly, to obtain a reference to the mover object. With that reference, they can ask the operating system for object migration operations.

7 Other distribution issues

Other distribution related issues, like replication or transactions, could be implemented in a similar way. The usual message passing from the invoking object to the invoked object could be redirected to a Transaction Processing Monitor by changing the behaviour of the sender meta-object. A persistence service will be also needed to make changes durable after transaction commit. Replication is not supported in the basic object model of the abstract machines, but could be provided by the operating system. A possible solution would be the modification of the sender or receiver meta-objects, to redirect invocation requests to the most appropriate replica.

8 Current implementation status

At the moment, several projects for endowing structural reflection to the abstract machine have been finished. One of them has implemented dynamic structural reflection, so classes can be modified at execution time (for instance, adding new methods or instructions) and objects of those classes get modified as well. A distribution prototype has been also finished, but with the distribution policies and mechanisms at the machine level. We are now working on a full reflective machine to meta-implement the distribution issues presented in this paper and other operating system services like persistence.

9 Conclusions

An Integral Object-Oriented System (IOOS) is a system based solely on the object-oriented paradigm, which provides a “world of objects” environment: a virtually infinite space where objects live indefinitely and exchange messages regardless of their location. The Oviedo3 is an IOOS based on an object-oriented abstract machine that provides a basic object model.

Distribution is one of the most important properties that must be present in these environments. Where to introduce distribution support in the IOOS is not trivial if we want to be able to adapt

(customize) distribution to specific needs. An immediate solution could be to modify the basic object model provided by the abstract machine with distribution properties. This is not a good solution, because mechanisms and policies will be fixed in the IOOS core, and could not be modified or replaced.

A better solution is to endow the abstract machine with a reflective architecture. The reflective architecture will divide the world of objects in two levels: objects at the basic level implement usual user applications. Objects at the meta-level are responsible of implementing usual operating system level services. The object-oriented paradigm is not broken and operating system services can be easily adapted by object modifying or replacement.

The default method invocation mechanism provided by the abstract machine is rewritten at the meta-level to make possible remote method invocations. A set of meta-objects implements the new method invocation mechanism overcoming the difficulties introduced by the possible dispersion of objects, classes and references involved.

Object migration is also solved at the meta-level in a transparent way, so objects been moved are unaware of this fact. Another set of meta-objects is responsible of stopping any internal activity in the object, obtaining its encapsulated state in a string, sending the string in a message to the chosen destination and reactivating the object there. Several policies could be used to decide when, where and how much to move.

While preserving transparency, this reflective architecture is flexible enough as to also permit special applications to control distribution issues by means of a set of meta-objects exposed for this purpose.

References

[ATA+97] D. Álvarez Gutiérrez, L. Tajes Martínez, F. Álvarez García, M. A. Díaz Fondón, R. Izquierdo Castanedo, and J. M. Cueva Lovelle. “An object-oriented abstract machine as the substrate for an object-oriented operating system”. In ECOOP’97 Workshop in Object Orientation in Operating Systems, June 1997.

[DLA+91] P. Dasgupta, R.J. LeBlanc, M. Ahamad y U. Ramachandran. “The Clouds Distributed Operating System”. IEEE Computer, 24(11). 1991. Pág. 34-44.

[Jul88] Eric Jul. “Object Mobility in a Distributed Object-Oriented System”. PhD Thesis. Department of Computer Science, University of Washington, Seattle, Washington, December 1988

[Mae87] P. Maes. “Concepts and Experiments in Computational Reflection”. In Proceedings of the 1987 OOPSLA, pp. 147-155. 1987.

[OMG99] Object Management Group. “The Common Object Request Broker: Architecture and Specification, revision 2.3”. Object Management Group. Junio de 1999. Disponible en URL: http://www.omg.org.

[TAD+98] L.Tajes Martínez, F. Álvarez García, M.A. Díaz Fondón, D. Álvarez Gutiérrez y J.M. Cueva Lovelle. “A Computational model for a Distributed Object-Oriented Operating System Based on a Reflective Abstract Machine”. In ECOOP’98, Workshop on Reflective Object-Oriented Programming and Systems.