



*Rachid Guerraoui and Mohamed E. Fayad*

# OO Distributed Programming Is *Not* Distributed OO Programming

**O**bject-oriented distributed programming is fundamentally different from building mechanisms that hide remote invocations behind traditional, centralized abstractions. The metaphor of a community of independent objects communicating by passing messages is misleading and dangerous when thinking in terms of a distributed system. That metaphor should not hide the significant difference between distributed interaction and local interaction.

## **Magic Acronyms**

The acronyms CORBA, DCOM and RMI have two common characteristics. First, they sound very “in” and current. They are particularly attractive in crucial meetings about the future of enterprise computing. A proposal for a funding agency might have a better chance of success if it has some of these acronyms in its introduction. Marketing professionals frequently advise: “When you talk to a customer, use a lot of acronyms and when you find one that the customer does not

seem to understand, keep using that one; this will clearly establish the roles in the discussion and improve your chances of making a good deal.” Second, these acronyms are all the fruits of a big wedding, a wedding held every 10 years or so.

This case is the wedding of two old, but still exciting, technologies—OO programming and distributed systems. OO programming is a tool for improving productivity, quality and innovation in software development. Objects are collections of operations sharing a state that provide persistent services over time. Operations are described through an interface providing information about attributes and actions visible to clients. The implementation of each object is encapsulated—hidden from clients of the object.

A distributed system consists of several computers working together. Some of the negative characteristics of a distributed system are: independent failure (some computers may break while others keep going); unreliable communication in which

messages may be lost or garbled; insecure communication, unauthorized eavesdropping and message modification; and finally costly communication; interconnections among a group of computers usually provide lower bandwidth, higher latency, and higher cost communication than that available among the independent processes within a single machine. However, the union of these technologies is a winning combination for the 1990s, comparable to the combination of graphical interfaces and relational databases that was very successfully exploited in client-server architectures at the end of the 1980s. The distributed objects paradigm is usually considered a further step towards open computing. The target architectures are no longer closed and centralized with the roles of clients and servers established in advance, but open architectures in which objects alternately play the role of clients and servers, cooperating, in a democratic way, through the Internet. The components of an open system are

**Distribution transparency is impossible to achieve in practice. Precisely because of that impossibility, it is dangerous to provide the illusion of transparency.**

not defined in advance, but are discovered and added during program execution.

One might wonder whether the marriage between object and distribution is not an arranged union, designed to expand the market share of the industries behind each of those technologies, and this may be a part of the truth. However, it does not take much thinking to come to the conclusion that there are some objective reasons to believe that merging the concepts of object and distribution is indeed motivated by more technical reasons. One reason, proposed for the past couple of decades, is that objects are very good candidates for modeling units of distribution because they encapsulate data and procedures. Also, OO programming is considered a good method for the development of complex systems, and a distributed system is just another example of a complex system.

Just as encapsulation and inheritance were very necessary in the successful building of graphical user interface environments, modularity, extensibility and reuse, those bastions of OO programming, could be of great

help in building distributed systems.

### **Once Upon a Time**

Before discussing those motivations and what could and could not come of them, let us first review the history of the union.

The first OO language, Simula, developed in 1965 by Dahl and Nygaard in Norway, was basically aimed towards simulation applications. However, the designers of Simula realized that the concepts underlying the language could be applied to other kinds of applications. They developed Simula's successor, Simula 67, by extending the Algol 60 language with two fundamental concepts, encapsulation and inheritance, and two companion mechanisms, instantiation and subclassing. The concept of encapsulation, gathering data and procedures inside the same anthropomorphic entity, (object), turned out to be particularly powerful. Encapsulation entails building the components of a system in a modular way while hiding the details of their implementation. The concept inspired several new developments, notably abstract data types, developed by Hoare in

1972 and the design of the CLU language by Liskov and Snyder, circa 1977. The concept of inheritance was particularly useful for improving the readability of programs and prototyping. Kay and Goldberg exploited the power of the concept through the design of the Smalltalk library of graphical interfaces. The term "object-oriented language" was associated with languages like Simula, Smalltalk and Eiffel, which supported mechanisms of instantiation and subclassing.

Until the success of C++, a more pragmatic language closer to the machine code than its predecessors, these languages were considered academic toys, at best useful for the composition of GUI windows. Simula 67 already provided some support to simulate logically distributed programs. Through the notion of co-routines, the programmer could describe independent entities and run them in a concurrent fashion. This was quite natural since Simula was targeted at simulation applications and many such applications are inherently concurrent and distributed (such as process control applications). Moreover, the object concept was aimed at modeling autonomous (concurrent) entities. Probably because of the cultural inheritance of the sequential programming culture and technological and speed limitations of network communication, the distribution aspect was not fully considered. In the late 1970s, Hewitt and Agha described a model of dynamic autonomous objects that communicate in an asynchronous way. These objects were called

actors. Around the same period, Liskov and Sheifler extended the CLU language to handle distribution: the result was Argus. In the meantime, Black, Hutchinson and Jul came out with a language and a system called Emerald, in which objects are mobile entities that can migrate from one machine to another (sound familiar?).

### **A Dangerous Metaphor**

Many developers of OO software describe their programs as sets of independent objects communicating through passing messages. When designing the Smalltalk system, for instance (a very nice OO program actually), Kay gave the image of objects as “little computers that communicate together.” One can easily bridge the gap between this view and a sensible representation of a distributed system. More precisely, it is very legitimate to view the “independent objects communicating through message passing” metaphor as an adequate one for modeling distributed programs. A set of objects communicating by passing messages looks very much like a distributed system. The anthropomorphic metaphor of objects as autonomous individuals, and messages as the communication paradigm between those individuals, matches very closely the structure and behavior of a distributed system.

*Autonomous entities communicating by passing messages.* The next step is to claim that the same programming paradigm can be applied both for centralized (single address space, single machine) programs and for distributed programs. In other words, one can take a program developed

without distribution in mind and execute it in a distributed system simply by placing the objects on different machines and providing a mechanism that transparently transforms local communication into distributed communication. The ultimate goal is to make distribution transparent to the programmer by hiding it through underlying inter-object communication. At the present time, most OO distributed systems provide some degree of distribution transparency (Roughly speaking, distribution transparency means the very fact of hiding distribution-related aspects.) Distribution transparency even became a metric for evaluating distributed systems: it is frequently heard or read that “system X is really great because, with X, one can view a remote object in the same way as a local object. As a consequence, one can directly reuse, in a distributed context, an application written in a centralized context (without distribution in mind).” This is a very misleading approach because, as we will discuss, distribution transparency is impossible to achieve in practice. Precisely because of that impossibility, it is dangerous to provide the illusion of transparency.

*The myth of transparent distribution.* Considering objects to be the units of distribution of a program is indeed sensible. After all, an object is a self-contained entity that holds data and procedures and that is supposed to have a uniquely identified name. Nevertheless, providing a mechanism that can transparently transform local communication into distributed communication is not very reasonable.

A transparent transformation first means encoding and decoding, behind the scenes, the name of an operation and its arguments into some standard format that could be shipped over a wire. The overhead introduced by this task varies according to the degree to which the marshaling process is dynamic (for example, static versus dynamic stubs and skeletons) and the method used to code the data to be transferred over the wire (usually called serialization). In any case, the time taken to perform this task is strictly overhead in comparison with the latency of a local invocation. After the marshaling task is over, the caller's thread is typically blocked until a reply is returned. At the remote machine, a dual unmarshaling task is triggered and the operation of the callee is invoked. When this operation terminates, the reply is marshaled and then sent back over the wire to the caller. Latency might become a major concern here because transparency can only be effective if the time taken by a remote invocation is similar to that of a local invocation. One might argue that the generation of static stubs and skeletons together with optimized serialization techniques might lead to very good performance over a fast transmission network and indeed make a remote invocation look like a local one. This might indeed be true in a LAN under the very strong assumption that no process, machine, or communication failure occurs.

If a failure occurs somewhere between the caller and the callee (for instance, the link, the callee process, or the callee machine fails), then it is impossible to con-

tinue to provide any illusion of transparency, at least not with a simple remote method invocation protocol. Two more sophisticated techniques might be considered to extend the basic remote invocation mechanisms: retry behind the scenes or replication. The first technique requires some transactional mechanism that would abort the effects of the callee operation execution in case of a failure and retry the invocation until it eventually succeeds. The second alternative consists in masking failures through replication. Neither of those techniques can hide failures for all types of objects in all kinds of failure scenarios. Even when they can, the overhead introduced by the mechanisms is usually considerable and makes a significant difference between the latency of a remote invocation and that of a local one.

### **Distribution Awareness**

Being aware of the fundamental difference between distributed object invocation and local object invocation has the merit of giving up the myth of distribution transparency and the claim that an OO concurrent language is enough because distribution is only an implementation detail. However, taking the difference seriously is not straightforward.

One way of making distribution explicit is by extending an OO language with specific keywords to identify objects that need to be accessed remotely and for which specific mechanisms are deployed. For example, the programmer would explicitly mark the objects that need to be accessed remotely and develop specific exception handlers to deal with failed invocations. For

remote objects, the compiler would then automatically generate specific mechanisms that guarantee some form of all-or-nothing invocation semantics. The flexibility of this approach is very limited precisely because it is hard to predict which distribution mechanisms should best fit the programmer's application. Again, one could enrich the language with keywords to identify the mechanism required for each category of objects. For instance, immutable objects can easily be replicated without the need for any distributed synchronization (In other words, no distributed protocol among replicas is needed to be performed in order to ensure that the replicas behave as a single non-replicated object).

Furthermore, objects with commutative operations would require only some form of causal ordering among replicas. This approach would lead to a significantly complex language and would anyway be limited in its scope since one cannot predict the mechanisms that could be desirable in every scenario: asynchronous versus synchronous, asynchronous with future versus one-way, replicated invocation versus non-replicated, transactional versus non-transactional, and so forth.

It is often argued that an approach based on reflection and separation of concerns provides the adequate degree of flexibility required in distributed programming. It is true indeed that the ability of a system to describe its basic characteristics in terms of the system itself makes it much easier to extend. However, it is not clear whether one can factor out the distribution-related aspects of an object outside that object. The functionality of the

object has direct impact on its distributed implementation.

Moreover, reflection does not provide a way per se to identify the set of basic mechanisms on which to rely when building a distributed mechanism. In other words, reflection is indeed an effective way to customize and plug various mechanisms in an existing system but does not really help in building those mechanisms. To make an analogy, reflection can be used to customize the presentation of an object by plugging the right GUI mechanism for every object, but does not really help in building and structuring the GUI mechanisms themselves.

OO DISTRIBUTED PROGRAMMING should not be about hiding distribution behind classical abstractions of traditional (centralized) OO languages. In other words, OO distributed programming should not be about building a distributed OO language or system. A future column will explain that OO distributed programming is about viewing distribution as the application domain from which fundamental abstractions should be extracted and classified. This is a challenging area of research and taking it seriously requires going beyond wrapping existing technologies with OO-like interfaces. **C**

---

**RACHID GUERRAOUI** (Rachid.Guerraoui@epfl.ch) is an assistant professor in the Computer Science department at the Swiss Federal Institute of Technology, Lausanne (EPFL), Switzerland.

**MOHAMED E. FAYAD** (fayad@cs.unr.edu) is an associate professor in the Computer Science department at the University of Nevada, Reno.

---

© 1999 ACM 0002-0782/99/0400 \$5.00