What object oriented distributed programming does not have to be, and what it may be

Rachid Guerraoui[1]

Swiss Federal Institute of Technology
1015 Lausanne, Switzerland
Rachid.Guerraoui@epfl.ch

Object oriented distributed programming is not distributed object oriented programming and is also more than wrapping existing distributed systems with object oriented dressing. Object oriented distributed programming is rather about identifying and classifying the basic abstractions underlying distributed computing. Identifying those abstractions and building a distributed operating system around those abstractions is an open and challenging area of research.

1. Talking to a customer

"When you talk to a customer, make use of a lot of acronyms and when you find one that the customer does not seem to understand, keep using that one; this would clearly establish the roles in the discussion and improve your chances of making a good deal". This is one of the advises I remember from a marketing teacher of mine.

CORBA, DCOM and RMI are acronyms that you have most probably heard about. You might have felt uncomfortable with the frustrating impression being the only one not to understand them in a meeting, or worst, you might have used them to impress your managers and look on the wave too. Those *magic* words have two common characteristics. *First*, they sound very *in* and *sexy* and are particularly appreciated in hot meetings about the future of the computing technologies. Writing a proposal for a DARPA or an Esprit project might have better chances of success if it has some of those acronyms in its introduction. *Second*, these acronyms are all the fruits of a big wedding; one of those weddings that are held every ten years or so. The wedding of two old but still exciting technologies: object oriented programming and distributed systems. The union of these technologies is a winning combination for the 90's; comparable to the combination of graphical interfaces and relational databases that was very successfully exploited in client-server architectures at the end of the 80’s.

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 ones where objects alternately play the role of clients and servers, cooperating, in a democratic way, through the Internet. The components of an open system are not defined in advance, but rather added and discovered during program execution.

[1] Currently with Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304.

Big companies are investing considerable effort in making the technology of *distributed objects* viable from an economical point of view. The paradox is that the researchers, especially academic ones with small budgets, might see their role merely confined in evaluating technologies and trying to catch up what industry is coming with. Fortunately for the research community, and unfortunately for the users of the technology, the union of object and distribution is far from being successfully achieved. Beside what is claimed in many flyers and web pages, Interesting problems are still to be solved, behind the strategic competition involving CORBA, DCOM and Jini.

2. A marriage of convenience ?

One might wonder whether the marriage between *object* and *distribution* is not just an arranged union, designed to expand the market share of the industries behind each of those technologies. This may be a part of the truth. However, it does not take a lot of 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.

• The new operating system

One reason for the good intention of the wedding, and this reason was advocated for the last couple of decades, is that objects are very good candidates for modeling units of distribution because they encapsulate data and procedures. Distributed objects are the adequate support to build the operating system of the new computer, that is, the *network*. Instead of building a huge operating system aimed at answering all requests in all possible configurations (e.g., a Windows NT like system with 40.000.000 lines of code for handling all situations), the idea behind distributed objects is to view the operating system as an extensible set of small and complementary components (objects). Each object is a partial operating system, responsible for a specific task and able to communicate and require services from other objects. One would start with a small set of objects and extend this set according to the application needs by using other objects from the network. Before discussing this motivation and what could and could not come out of it, let us first review the history of the union.

• A brief history of the union

The first object oriented language, *Simula*, developed in 1965 by O-J Dahl and K. Nygaard in Norway, was basically aimed towards simulation applications. However, the designers of the language realized that the concepts underlying the language could be applied to other kinds of applications. They developed the successor of Simula, 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, called an *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, by C. Hoare, in the early 70's, and the design of the CLU language by B. Liskov and A. Snyder, few years later. The concept of inheritance was particularly useful for improving the readability of programs and rapid prototyping. A. Kay and A. 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 Simula, Smalltalk and Eiffel, which supported mechanisms of instantiation and subclassing. Until the success of C++, more pragmatic and closer to the machine code than its predecessors, object oriented 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 *coroutines*, the programmer could describe independent entities and make them run in a concurrent fashion. This was quite natural, after all, since Simula was targeted at simulation applications and many such applications are inherently concurrent and distributed (e.g., process control applications). Moreover, the object concept was aimed at modeling autonomous (i.e., concurrent) entities. Probably because of the cultural inheritance of the sequential programming culture and technological limitations of the speeds of network communication, the distribution aspect was not fully considered. In the late 70's however, C. Hewitt and G. Agha described a model of dynamic autonomous objects that communicate in an asynchronous way. These objects were called *actors*. Around the same period, B. Liskov and R. Sheifler extended the CLU language to handle distribution: the result was the *Argus* language. In the meantime, A. Black, N. Hutchinson and E. Jul came out with a language and a system called *Emerald*, where objects are mobile entities that can migrate from one machine to another (sounds familiar, doesn't it?).

3. The myth of distribution transparency

Many developers of object oriented software describe their programs as sets of *independent objects communicating through messages passing*. When designing the Smalltalk system for instance (a very nice object oriented program actually), Alain Kay has given 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 through *message passing* looks very much like a distributed system. The anthropomorphic metaphor of objects as autonomous individuals, and messages as the communication paradigm between those individuals, match very closely the structure and behavior of a distributed system.

• Autonomous entities communicating through message passing

The next immediate 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 under inter-object communication and provide the illusion that everything happens just like in a centralized system. Most so called *object oriented distributed systems* that have been developed so far provide some degree of distribution transparency. Distribution transparency even became a metric for evaluating distributed systems: it is frequently heard or read that *"system X is better than system Y because X provides full distribution transparency whereas Y does not"*..

Considering objects to be the units of the distribution of a program, is indeed sensible. After all, an object is a self-contained entity that holds data and procedures and is supposed to have a uniquely identified name. Nevertheless, providing a mechanism that *transparently* transform local communication into distributed communication is not very reasonable. As we will discuss below, complete distribution transparency is impossible to achieve in practice. Precisely because of that impossibility, it is dangerous to provide the illusion of transparency.

• Coding/encoding behind the scene

Transforming a local invocation into a remote one first means encoding and decoding, behind the scene, the name of an operation and its arguments into some standard format that could be shipped over the wire. The overhead introduced by this task varies according to the degree to which the marshalling process is dynamic (i.e., static vs. dynamic stubs and skeletons) and the method used to code the data to be transferred over the wire (usually called serialization technique). In any case, the time taken to perform this task is a strict overhead in comparison with the latency of a local invocation. After the coding task is over, the client's thread is typically blocked until a reply is returned. At the remote machine, a dual decoding task is triggered and the operation of the server is invoked. When this operation terminates, the reply is marshaled and sent back over the wire to the caller. Latency might become of 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 performances 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 [Waldo et al. 94].

• Masking failures

If a failure occurs somewhere between the client and the server (i.e., the link, the server process or the server machine fails), then it is just impossible to provide any illusion of distribution transparency, at least not with a simple remote method invocation protocol. More sophisticated, techniques should be considered to extend the basic remote invocation mechanisms: *retry behind the scene* or *replication*. The *first* technique requires some transactional mechanism that would abort the effects of the client operation execution in case of a failure, and then retry the invocation until it eventually succeeds. The *second* technique consists in masking failures through replication. None of those techniques can hide failures for all types of objects in all kinds of failure scenarios [Vogels et al. 98]. Even in the situations where they can, the overhead introduced by the mechanisms is usually considerable and makes a significant difference between the latency of a remote invocation and the latency of a local one.

• Marking distributed objects

Being aware of the very fundamental difference between remote object invocation and local object invocation has the merit of giving up the myth of *distribution transparency* and the claim that an object oriented concurrent language is enough because distribution is just an implementation detail. However, going a step further and taking the difference seriously is not straightforward.

One way of making distribution explicit is by extending an object oriented 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 degree of flexibility of this approach is very limited precisely because it is hard to predict which distribution mechanisms should best fit a given 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*. No distributed protocol among replicas is needed in order to ensure that the replicas of an immutable object 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 as 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, etc [Briot et al. 98]

- **Separation of concerns**

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 a lot easier to extend. However, it is not clear whether one can factor out the distribution-related aspects of an object and express them outside that object. The functionality of the object has direct impact on its distributed implementation. Moreover, reflection does not per se provide a way to identify the set of basic mechanisms to rely on 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.

4. Back to the future

There are as many definitions of the object oriented programming paradigm as there are papers trying to define the paradigm. However, if we go back to the roots, to the Simula language, we can state that object oriented programming is, first of all, about mapping a physical model of a problem domain into a program.

An object oriented program is aimed at reflecting the structure of the application domain through a one-to-one correspondence between objects of the application domain and objects of the computational model [Guerraoui 96]. Bjarne Stroustrup defined object oriented programming as follows: "*decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance*". Where there is no such commonality, data abstraction suffices *("decide which type you want and provide a full set of operations for each type"*) [Stroustrup 87].

If the application domain is the administration of a university for example, then the actual students and university employees are represented by objects, and the concepts of student, graduate student, teaching assistant, professor, etc, are represented by classes of an inheritance hierarchy. This modeling based approach improves the productivity and quality of software development together with various forms of reuse both at the level of an individual class and at the level of a group of classes, i.e., a library or a framework [Madsen 88]. An application domain for which object orientation has been particularly successful is that of graphical interfaces. Following an object oriented programming approach, concepts like window, mouse, and button are represented by object classes. A distributed system is just another example of an application domain. In other words, a distributed system can be viewed as the application domain for which we should identify and classify the fundamental abstractions. Following this approach in a rigorous way is challenging. It means structuring a distributed system in the form of a class library all the way down. It also means going beyond current practices, which consist in wrapping built-in toolkits with object oriented like dressing.

• The outfit does not make the monk

Since object orientation has become a respectable denomination in the industrial arena, there has been an important tendency to wrap built-in prototypes or products behind object oriented interfaces. This dressing is usually accompanied with a marketing campaign advertising the benefits of the new object oriented line of the system. This has been particularly true for various distributed programming toolkits that have been made CORBA compliant through a careful redesign of their interfaces in IDL (the Interface Description Language designed by the OMG [OMG]). While an object oriented interface tends to make the functionality of the distributed toolkit easier to understand, it does not fundamentally improve its modularity, extensibility, or the reuse of its components; all what object oriented programming is indeed about.

The question here is related to granularity. Most toolkit products for distributed programming are large monolithic entities. Wrapping them behind an object oriented like interface and calling them services (versus toolkits) does not make them object oriented. To make this point of view more concrete, consider three examples of *services*, the interfaces of which have been (or are in the process of being) standardized by the

OMG: the CORBA transaction service, the CORBA event service and the future CORBA service aimed to achieve fault-tolerance through entity redundancy. The transactional service is described as a set of interfaces for transaction manipulation and atomic commitment. In particular, the transactional service provides an interface for a two-phase commit protocol. The event service is a set of interfaces for event manipulation along a publish/subscribe communication pattern. A set of consumers can subscribe to a channel and subsequently receive, through an asynchronous multicast, some of the events put into the channel. Finally, the fault-tolerant service is based on the idea of entity redundancy through some notion of group. Roughly speaking, the failure of an object is hidden by the group.

The interfaces specified (or under specification) by the OMG are aimed at describing the general functionality of each of the services. The services are considered separately and their underlying common components are not captured. An object oriented design of the underlying problem domain would have led to capturing and factoring out common underlying concepts such as persistence, failure detection, multicast and distributed agreement. Furthermore, given the large grain level at which the interfaces are provided, it is not clear at all how one could extend any of these services. Extensibility does not seem to have been a major concern when designing the interfaces. For example, it is all but straightforward to extend the CORBA transactional service and provide an alternative distributed commitment protocol to the two-phase commit protocol provided by default. There are many cases where other forms of commitment are more appropriate. Identifying a very generic notion of agreement that could be subclassed to provide different forms of distributed commitment would have been a more object oriented approach. In fact, such a generic notion of agreement could be used also for the fault-tolerance service, i.e., behind group membership, and for the event-service, i.e., to guarantee, when required, some form of *atomicity* or *globally ordered* view for multicast event delivery. Furthermore, there are many distributed applications that require transactional, fault-tolerance and publish/subscribe semantics. The fact that the services providing each of those semantics are considered separately can turn out to be a major handicap for the application developer who is willing to make use of any combination of those services. In contrast, searching for the commonalties underlying those services and factoring them out could indeed be of a major help for the developer.

The above observations are, by no way, critics of CORBA. The OMG has specified the most advanced distributed system infrastructure that turns out to be viable from an industrial perspective. The observations above mainly state that complying with CORBA specifications does not make a distributed service object oriented, i.e., complying with CORBA specifications does not mean that the service is more modular, extensible nor that its components are easier to share and reuse.

5. Abstraction vs. transparency

Taking object oriented distributed programming seriously means structuring a distributed system in the form of a set of first class objects all the way down. Designing high level distributed frameworks should go first through identifying the basic abstractions, and second through building libraries that help to understand those abstractions and their interactions. These steps cannot be achieved without a deep understanding of the abstraction domain, that is, *"distributed computing"*. The most fundamental questions in designing basic abstractions are first technical questions [Meyer 96]. Classification and specialization mechanisms, as offered by object oriented languages, are then appropriate to organize a library/hierarchy of higher level abstractions.

• First class citizens

To make the claim more concrete, consider the example of a failure detection mechanism. Quoting L. Lamport, "*a distributed system is also one that stops you from having any work done because of the crash of a machine you have never heard about*". The notion of *partial failure* is a fundamental characteristic of a distributed system: at a given time, some components of the system might have failed whereas others might be operational. The ability to hide partial failures or recover from them is a crucial metric for measuring the reliability of a distributed system. All reliability schemes that we know make use of some form of *failure detection* mechanism. Failure detection is a crucial component in transaction processing, replication management, load balancing, distributed garbage collection, as well as in applications that require monitoring facilities like supervision and control systems.

In most distributed systems however, failure detection is either completely left to the application developer or completely hidden. Failures are handled through mechanisms like exceptions and it is up to the programmer to distinguish a physical failure (the crash of a machine) from a logical failure specific to the application's semantics. Some reliable distributed toolkits like transaction monitors and group communication systems provide some support for failure detection through *timeouts*, e.g., an object is suspected to be faulty if it does not respond to an invocation after some time. The specific code that handles timeouts however is usually mixed up with the code of the distributed protocols in charge of failure hiding or failure recovery. For instance, in transaction monitors, the code for timeout management is usually mixed up with the code for distributed transaction manipulation protocols like atomic commitment. It is very difficult, if not impossible, to adapt the failure detection mechanism to the network topology without modifying the application or the underlying distributed protocols. The only parameters that are generally left to the developer are the timeout values. These are indeed important parameters that enable the developer to trade latency (short timeouts and hence fast reaction to failures) with accuracy (long timeouts and hence more accurate failure detection). Nevertheless, the developer cannot however tune the failure detection protocol itself according to the network topology. This can be viewed as a serious drawback of existing distributed systems and can reduce their scalability and more generally their applicability in various contexts. For example, according to the network topology and the communication pattern of the application, the choice between a *push* (*heartbeat*) or a *pull* (*are-you-alive ?*) monitoring model can have an important impact on the performance of the system. In a push model, every component of the system is supposed to regularly send heartbeat information to the other components: a component is considered faulty if its heartbeats are not received by the other components in time. In a pull model, a component A monitors a component B by sending it *"are you alive ?"* messages. If B fails to respond to such a message after some timeout, A considers B to be faulty. None of the push or the pull model fits all situations. In a large scale system, one might use either of those models or even mix them to reduce the number of messages exchanged in the network.

Following an object oriented modeling style, failure detectors should be considered as first class citizens. That is, failure detection should be viewed as an abstraction, the complexity of which is encapsulated behind well-defined interfaces. These interfaces should be arranged in a hierarchy that provides different views of the service and different interaction paradigms for failure detection. In particular, the hierarchy may include specialized interfaces for the *push* and the *pull* execution styles. A *dual* monitoring model where the advantages of both styles are combined can simply be obtained by inheriting from both push and pull interfaces. On one hand, failure detection mechanisms should be separated from other mechanisms in the system to provide for better modularity and extensibility and, on the other hand, the entities being monitored should be abstract objects in the system to eliminate the mismatch between the need for failure detection at the level of application objects and the support provided by some operating systems to detect host failures. One can configure the failure detection service in such a way that the monitored units can range from specific application objects, processes, machines, or even sub-nets.

• Putting object oriented distributed programming to work

A Distributed system is several computers working together. Some of the characteristics of a distributed system are: *independent failure*, some computers may break whereas others keep going, *unreliable communication*, 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 between the independent processes within a single machine.

Finding out the right way to represent distribution-related concepts *(message, machine, process, failure detection, agreement, etc)* as first class programming entities, and then classifying them within hierarchical libraries and frameworks are indeed challenging issues. Higher level abstractions, such as various forms of remote method invocations and distributed event processing, could then be built out of such basic abstractions. A programmer can use a high level abstraction of a remote method invocation if it fits the requirements of the application. but nothing should prevent him or her from directly using the more fundamental abstractions and manipulating them as first class entities. Furthermore, following the Hollywood principle *"don't call us, we will call you"*, parts of the abstractions should be left to be customized by the application programmer, as one rarely finds, in any given library, the abstraction that is exactly needed in the application.

There are many cases where some form of message passing or event-oriented communication is more appropriate than remote method invocation. One can, indeed, build an event-oriented communication system using threads on top of a remote method invocation mechanism but this is, by no means, a natural construction and would lead to considerable overhead. Instead, the more basic abstractions of *messages and communication port* should be made available as first class objects. Similarly, there are many cases where specific entities of an application are best represented by *active objects (with its own thread of control).* One can certainly build an abstraction of an active object model in a concurrent language but nothing should prevent the programmer from using the more fundamental abstractions like *threads* and *semaphores*. Thanks to its well defined interface (wait and signal operations) and its well known behavior (train metaphor), the *semaphore* represents now a standard abstraction for concurrent programming, just as *arrays* and *sets* are basic abstractions for manipulating data structures in sequential programming. These abstractions came out of a mixed effort between theory and practice and are the heart of higher level abstractions and frameworks.

6. Summary

This paper argues that object oriented distributed programming should not be about wrapping existing technologies with object oriented like interfaces nor should it be about hiding distribution behind classical abstractions of traditional (centralized) object oriented languages. Roughly speaking, object oriented distributed programming has very little to do with building a distributed object oriented language or system. The metaphor of a community of *independent objects communicating through message passing* should not hide the fact that distributed interaction *is not* local interaction.

This paper claims that object oriented distributed programming is about viewing distribution as the application domain from which fundamental abstractions should be extracted and classified. Just as encapsulation and inheritance were very necessary in the successful building of graphical user interface environments, they could be of great help in building distributed operating systems.

7. Acknowledgments

The thesis of this paper, namely that there is more to object oriented distributed programming than building a distributed object oriented language or system, came out of a dinner with J. Edwards, B. Orfali and R. Session. Discussions with A. Black, J-P Briot, P. Felber, B. Garbinato, J. Gray, S. Frolund, D. Lea, R. Resnick and J. Sventek provided additional insights to the picture.

8. References

[Briot et al. 98] J-P Briot, R. Guerraoui and K-P Lohr. *Concurrency and Distribution in Object oriented Programming.* ACM Computing Surveys, September 1998.

[Guerraoui 96] R. Guerraoui et al. *Strategic Research Directions in Object oriented Programming*. ACM Computing Surveys, 28(40), December 1996.

[OMG] The Object Management Group. http://www.omg.org.

[Madsen 88] O. Lehrmann Madsen and B. Moller-Pedersen. *What object oriented programming may be and what it does not have to be*. Proceedings of the European Conference on Object oriented Programming, Springer Verlag, LNCS 322, 1-20, August 1988.

[Meyer 96] B. Meyer. *The Reusability Challenge.* IEEE Computer. 1996.

[Stroustrup 87] B. Stroustrup. *What is "Object Oriented Programming"* ?. Proceedings of the European Conference on Object oriented Programming, Springer Verlag, LNCS 276, 51-70, June 1987.

[Vogels et al. 98] W. Vogels, R. van Renessee and K. Birman. *Six Misconceptions about Reliable Distributed Computing*. Proceedings of the Eight ACM SIGOPS European Workshop, September 1998.

[Waldo et al. 94] J. Waldo, G. Wyant, A. Wollrath and S. Kendall. *A Note on Distributed Computing*. Technical Report, Sun Microsystems Laboratories, Inc, November 1994.

About the author

Rachid Guerraoui obtained a M.S. in computer science from the University of Paris and a M.S. in electrical engineering from Ecole Superieure d'Informatique, Electronique et Automatique (Paris), both in 1989. From 1988 to 1992, Rachid Guerraoui worked at the Centre de Recherche de l'Ecole des Mines de Paris and then at the French Atomic Agency in Saclay. In 1992, he obtained a Ph.D in computer science from the University of Orsay (Paris) and joined the Computer Science Department of the Swiss Federal Institute of Technology in Lausanne (EPFL) where he holds a position of Assistant Professor. He is currently on leave as a Faculty Hire at Hewlett-Packard Laboratories in Palo Alto, California. Beside triathlons, Rachid is interested in object oriented programming and distributed systems.