Position paper

The Object Paradigm is to be reconsidered for Distributed Systems

Antonio Corradi, Letizia Leonardi

Dipartimento di Elettronica, Informatica e Sistemistica - Universita' di Bologna
2, Viale Risorgimento - 40136 Bologna - ITALY
Ph: +39-51-6443001
e-mail: BOARI@bodeis.cineca.it

Introduction and Motivations

The **object paradigm** has been proposed as a break-through because of many of its features: information hiding, dynamic object communication, grouping of information into classes, inheritance among classes, possibility of changing behavior at run-time by manipulating classes.

The **information hiding** property states a clean distinction between external visibility and internal perspective of an object. The external world has reduced visibility of the inside of an object: normally only an **interface** constitued by operation names is visible. Within any object, instead, there is the full visibility, in particular of the object state. This makes possible that each object decides its internal policies independently of any environment decision.
Object communications produce the global computation in object-based systems. An object can communicate with another if and only if knows it via a reference [Lis79]. In object systems, a communication implies that client objects request operations defined in the interface of server objects. These requests are exchanged by using message-passing mechanisms. **Dynamicity** of communications means that an object can change its visibility, i.e. the objects it can refer.
Classification and factorization by **inheritance** gives object environments the possibility of applying reusability and extensibility. On the one hand, object applications can be designed and programmed by reusing already developed software. On the other hand, applications based on objects can evolve to follow changes of specification. In the case of dynamic changes, the possibility of updating classes at run-time constitutes an useful tool.

The above described properties have oriented object-based systems toward rapid prototyping of applications [GolR83]. Only recently, more protected perspectives emerged and suggested new trends in the sense of protection [Mey88].

We think that a full exploitation of the potential of the object paradigm can only derive from the exploration of the issues neglected by its first implementations. In particular considering distributed targets, the areas of intervention span from the provision of concurrency to fault tolerance and persistency strategies.
Instead of concentrating on peculiar implementations of object systems to face specific problems, we feel that it is time of reconsidering the general properties of the object paradigm. The aim is to identify the extensions of the object paradigm in addressing the issues distributed systems propose:

- concurrency;
- communication;
- replication;
- dissemination;
- persistency.

Concurrency aspects

The structuring of an application into objects can simplify the phase of identifying **parallelism** [CorL90]. In fact, each object can constitute not only the structuring unit, but also the **processing unit.**
According to this point of view, active object systems associates a thread of execution with each object. This introduces the first dimension of parallelism into the object paradigm: we call it **inter-object parallelism.** The object thread is generally represented by one heavyweight process: after all, objects do not have an ephemeral life time.

The second dimension of parallelism is due to the possibility of associating with an object not only one execution thread but several ones. We call this second kind of parallelism **intra-object parallelism.** It introduces the need for synchronization (i.e. **scheduling**) of activities internal to a single object [CorL90]. Internal threads are generally mapped into lightweight processes because their creation/destruction is much faster than the object containing them.
The two components of parallelism have been first neglected in the first implementation of the object paradigm, but we think that they are a key toward the solution of the identification of parallelism.

Moreover, in a distributed architecture, the object can become not only the unit of parallelism, but also the **unit of allocation.** Once the objects of an application have been logically identified, the decision of their allocation to available processors is driven first by the total number of objects per processor and then by the internal concurrency to be obtained. These two levels in the allocation phase furnishes a guide when mapping objects to massively parallel architectures [CiaCL89].

Distribution also suggests the opportunity of several name spaces, instead of a unique one [Tsi87]. A distributed object paradigm can so be organized into **contexts**, that represent abstraction of physical processors, each one with its name space. Contexts contain objects and, conversely, objects belong to one context only.

Communication relationships

Although we consider distributed environments, object communication at the paradigm level must be **independent** of object location. A client must be not aware of where its server is allocated (and vice versa). This constitutes the basis for a complete **transparency**.

The first implementations of the object paradigm generally provides a **synchronous** mechanism for the communication between client and server objects [GolR83]: the client waits until the server yields a result. This mode of communication is too constrained to suit any problem of a distributed environment.

More asynchronous ways of communication can be considered (and have been), with and without the possibility of successively receiving a result. The asynchronous mechanism that allows the client to receive the result at a future time (called sometimes future communication) is the most flexible: any communication scheme can be implemented in terms of it.
Moreover, other different communication modes derive from the use of broadcast mechanisms directly provided at the architecture level.
Different communication modes help in providing a model reasonably close to the need of the application level. The risk that one must avoid is to propose mechanisms too pecualiar for a general usage. For example, negotiation protocols [DavL83], where several communications can occur back and forth from the client to the server before the server initiates the operation, are of too restricted usage and of such detail to be introduced at the paradigm level.

The client-server communication model describes only one interaction between two agents, but does not specify any mechanism to deal with a set of concurrent interactions.
The problem of grouping to gether multiple interactions is normally solved by transaction **serializability** [SchS84].
The importance of such mechanism for objects diminishes: if one wants to achieve the serialization of transaction semantics no additional mechanism needs to be introduced. It is only necessary to define a new object, the serializer, in charge of guaranteeing that interactions happen without inconsistency . It is the serializer that requests them to the involved objects.
The other aspect of transaction, **recoverability**, can be obtained by replication. Object systems tend to furnish transaction tools, implemented in this way [Delta4].
As a variation, the function of serialization can be sometimes carried out by the involved objects themselves, in a dedicated part [CorL90].

Replication model

The general object paradigm does not consider **replication**. Nevertheless, replication of objects seems a common strategy to obey the requirements of reliability and availability, depending on the set of faults one wants to tolerate [Bir85] [Delta4].
In fact, several proposals of distributed systems based on objects have already proposed solutions in the fault tolerance area by using replication.
Therefore, **object replication**, i.e. the recognition of copies as part of the same object, is a basic property that deserves to be embodied in the paradigm itself.

Object copies are either **passive** or **active** ones. Active copies execute all the same operations. Passive copies can be either on-line or on non volatile memory.
On-line back-up copies can be employed to reliably face faults in any resource, while several active copies of objects can be concurrently maintained to obtain availability.

When contexts are considered, one object is normally part of only one context. **Replicated objects**, instead, span contexts and have copies in more than one contexts. Replicated object copies must be recognized by the system as part of the same object.
Object replication takes advantage of information hiding: the policy of consistency between copies and the replication strategy, i.e. passive vs. active copies, are decided on an individual basis.
If one consider replication only for fault-tolerance, consistency among copies can be maintained following different schemes. These schemes depend on replication strategy, either active or passive: therefore, they can be either distributed agreement protocols or checkpoiting mechanisms [Delta4].

Nevertheless, object replication is not only tied to fault-tolerant purposes, but it has a more general meaning. In fact, in a distributed system, there is the need of recognizing a given service as a unique service. For example, a printing service is a general service that can be replicated in several contexts (i.e. nodes).
Since within each object there is a distinction between data and operations (normally contained in the object class), consistency can apply not only to data, but also to both parts.
The degree of consistency spans from tight consistency, that requires identical data and operations for all copies, to loose heterogeneity, when copies can evolve independently. Any intermediate shadow is allowed.
By using the printer example, tight consistency must be adopted in case the printers of each context are equal (and are to be maintained equal). Loose consistency, instead, accomodates the case of printers of different brands with different functions.
In an object environment, the provision of consistency protocols can take advantage of inheritance. Any object in need of a given protocol can derive it from a subclass of the one furnishing it.

Some aspects of dynamicity, in the object paradigm, are due to the presence at run-time of classes as first-class entities. This is on the one hand a constraint that imposes coresidence of objects and their classes in the same context, and, on the other hand, a sharing approach to be pursued in second generation distributed object systems.
Following the coresidence constraint, a class must be present in any context where its intances are. This implies that classes can be replicated objects with the problem of maintaining consistency. Since a variation of a class is less likely than a change of its instances, a reasonable consistency algorithm for classes can be the tight one.

Process group semantics can be implemented by using object replication. A taxonomy of group semantics can induce fruitful considerations when applied to the object model [Lia90].

Dissemination in time and in space

Object dissemination can be considered on the base of different dimensions: **in space** and **in time**.

Dissemination in space consists in **copying** objects within several contexts. Unlike replication, each copy is a new and autonomous object, not tied to the generator copy. Object copy obtains better communication performance the same as replication does. In fact, some communications that before the copy were remote can be transformed into local.

Dissemination in time, instead, means to **migrate** an object from one context to another. Object migration mimics process migration [Smi88], but present less problems than the process counterpart. This is due to the information confinement property of objects. Object migration can be used to effectively implement load sharing policies in a distributed system [Jul88].

Persistency issues

The possibility of using an approximation of stable memory guarantees a copy of information from which it can be restored after a crash. The current active copy saves its state on stable memory time by time.
Object systems must provide functions to interact with stable storage with its objects. One can freeze a copy of an object and recover information successively.

Objects can produce copies in non-volatile memory on individual basis. The client-server relationships between objects identifies hierarchies of objects: when a part of such a hierarchy is saved, other parts of hierarchy can be saved.
The recoverability aspect of atomicity can be implemented via the persistency feature.

A distributed object scenario

Summarizing, the object paradigm needs a limited amount of extensions to face the problems distributed systems must cope with.
The areas of extension of the paradigm are mainly:
- parallelism integrated with object properties;
- communication mechanisms that overcome the synchronous message passing limitations;
- objects distributed on different contexts;
- objects replicated in different contexts with possibly several consistency protocols;
- object dissemination by copying/migrating objects in/to different contexts;
- persistency of objects.

Transparency of allocation is a requirement not at the paradigm level but in the design of a programming environment. The paradigm works explicitly in the sense that any characteristic of objects such as allocation and replication must be explicitly specified. An implementation can make the user to follow a more implicit approach, by providing objects that incapsulate policies.
Part of the proposed extensions have been investigated by extending a Smalltalk system towards distribution [CorLZ90] and within the process of design of a support for objects on a massively parallel architecture [CiaCL90].

References

[Bir85] K.P. Birman: "Implementing Fault-tolerant Distributed Objects", IEEE Tr. on Software Engineering, v.SE-11, n.6, June 1985.

[CorL90] A. Corradi, L. Leonardi: "Parallelism in Object-oriented Languages", IEEE International Conference on Computer Languages, New Orleans (LA), March 1990.

[CorLZ90] A. Corradi, L. Leonardi, M. Zannini: "Distributed Environments based on Objects: Upgrading Smalltalk towards Distribution", IEEE Phoenix International Conference on Computer and Communications, Phoenix (AZ), March 1990.

[CiaCL89] A. Ciampolini, A. Corradi, L. Leonardi: "Objects on Massively Parallel Architectures", International Workshop on Supercomputing Tools for Science and Engineering, Pisa, Dec. 1989.

[CiaCL90] A. Ciampolini, A. Corradi, L. Leonardi: "Parallel Object Models in the Design of an Environment for Massively Parallel Architectures", ESPRIT Workshop of the Parallel Computing Action, Southampton, July 1990.

[DavS83] R. Davis, R.G. Smith: "Negotiation as a Methaphor for Distributed Problem Solving", Artificial Intelligence, v.20, 1983.

[Delta4] ed. D. Powell: "DELTA4: Overall System Specification", Technical ESPRIT Report, The Delta-4 Project Consortium, Nov. 1988.

[GolR83] A. Goldberg, J. Robson: "SMALLTALK-80: the Language and its Implementation", Addison-Wesley, 1983.

[Jul88] E. Jul et alii: "fine-grained Mobility in the Emerald System", ACM Trans. on Computer Systems, v. 6, n. 1, Feb. 1988.

[Lia90] L. Liang, et alii: "Process Groups and Group Communications: Classifications and Requirements", IEEE Computer, v.23, n.2, Feb. 1990.

[Mey88] B. Meyer: "Object-oriented Software Construction", Prentice Hall, 1988.

[SchS84] P.M. Schwarz, A.Z. Spector: "Synchronizing Shared Abstract Types", ACM Transactions on Computer Systems, v. 2, n. 3, Aug. 1984.

[Smi88] J.M. Smith: "A Survey of Process Migration Mechanisms", ACM Operating Systems Review, v. 22, n. 3, July 1988.

[Tsi87] D. Tsichritzis, et al.: "KNOs: KNowledge Acquisition, Dissemination, and Manipulation Objects", ACM Toplas, v.5, n.1, January 1987.