The "Ubiquitous Agent" Design Pattern

Jean-Marc Jézéquel*, Jean-Lin Pacherie†
I.R.I.S.A. Campus de Beaulieu
F-35042 Rennes Cedex, FRANCE
Tel: +33–99.84.71.92 — Fax: +33–99.84.71.71
E-mail: pacherie @irisa.fr jezequel@irisa.fr

June 17, 1996

Abstract

Encapsulating parallelism and synchronization code within object-oriented software components is a promising avenue towards mastering the complexity of the distributed memory supercomputer programming. However, in trying to give application programmers benefit of supercomputer power, the library designer generally resorts to low level parallel constructs, a time consuming and error prone process. To solve this problem we introduce a new design pattern called *Ubiquitous Agent*. An Ubiquitous Agent is an object that exists simultaneously on all processors involved in a distributed computation: it acts as a single ubiquitous entity capable of processing shared or distributed data in parallel. In this paper we describe the motivation, applicability and structure of this design pattern.

Keywords: Distributed and parallel processing

1 Intent

The concept of Ubiquitous Agent developed in this paper is inspired from both the G. Booch's classification of object relationships, and a reflection on the Iterator design pattern described in [2].

Booch's definition of agents [1] refers to a classification of object relationships that divides objects into *Actors*, *Servers* and *Agents*. Booch defines an agent as : "*An object that can both operate upon other objects and be operated upon by other objects; an agent is usually created to do some work on behalf of an*

*IRISA/CNRS, currently visiting Dept. of Info. Science, Tokyo University, Japan, under a JSPS grant

†IRISA/INRIA

actor or another agent". The main concept of Booch's definition is based on the notion of a service provider. An agent is an object that *does* something on a second object and on behalf of a third one, even if the target and the client of the service can be the same object.

The concept of Ubiquitous Agent we propose here is a pattern at the cross-road of the Iterators and the Booch's ones. The designer can use this pattern when s/he faces the problem of modeling and implementing the mapping of an operation over a collection of elements that might be shared or distributed across a number of processors.

2 Also Known As

Parallel Operator, Service Provider

3 Motivation

When designing our own application we faced the problem of modeling a mechanism that should behave as an agent from the programmer point of view and that can easily evolve to a distributed environment. The purpose of our research work is to provide a methodology to develop distributed applications by using parallel design patterns. These parallel design patterns can either be predefined or built as extensions of pre-existing patterns). This research activity led to the development and experimentation of the EPEE framework (more information about EPEE can be found in [4, 3]). An application oriented presentation of the Ubiquitous Agent design pattern is also available in [5] (in the context of N-body problems).

The Ubiquitous Agent design pattern we present here is based on the Iterator one and thus inherits from the same motivations:

- Provide a way to access elements stored in a container without exposing its internal structure.
- Separate the responsibilities of processing elements, providing elements, traversing container and storing elements between dedicated objects.
- Provide a way to implements polymorphic iteration so that a larger portion of client code can be reused.

But because we want to extend the Iterator pattern to enable a distributed execution of the applications designed using it, we extend the motivations with the following points:

- Extend the notion of iteration to model the behavior of an objet capable of providing a collection of elements. These elements can either be locally stored in containers or either be generated or also received through a communication channel.

- Provide abstract classes modeling distribution of elements over a set of processors. This concept is inherited from the data parallelism related technologies allowing in turn the distribution of control (as execution flow) over the processors using the *owner compute rule*.

- Provide technics to design an application first intended to run as a sequential program but that can also migrate to a distributed program using as easily as possible derivation technics based on multiple inheritance and polymorphism in the original code. This can safely be achieved under certain conditions, the purpose of the Ubiquitous Agent pattern is to constrain the programmer to satisfy these conditions when the pattern is correctly used.

4 Applicability

The use of the Ubiquitous Agent pattern is justified when:

- You want to access elementary objects and perform an operation on them without being dependent from a given memory representation of the object aggregating them (or absence thereof).

- You plan to run your application on a distributed system and want to benefit from the inherent parallelism existing between the operations performed over the elementary objects (or at least over several sub–sets of them) to speed it up.

5 Structure

See figure 1.

6 Participants

Agent (*deferred*): Defines an interface for performing a regular operation on a set of elements. The regular operation is defined in the deferred method *item_action (item : E)* and the way the elements are actually provided is masked in the *prov* attribute.

ConcreteAgent (*concrete*): Implements the regular operation to be perform on each elements it will process. This is a user defined object.

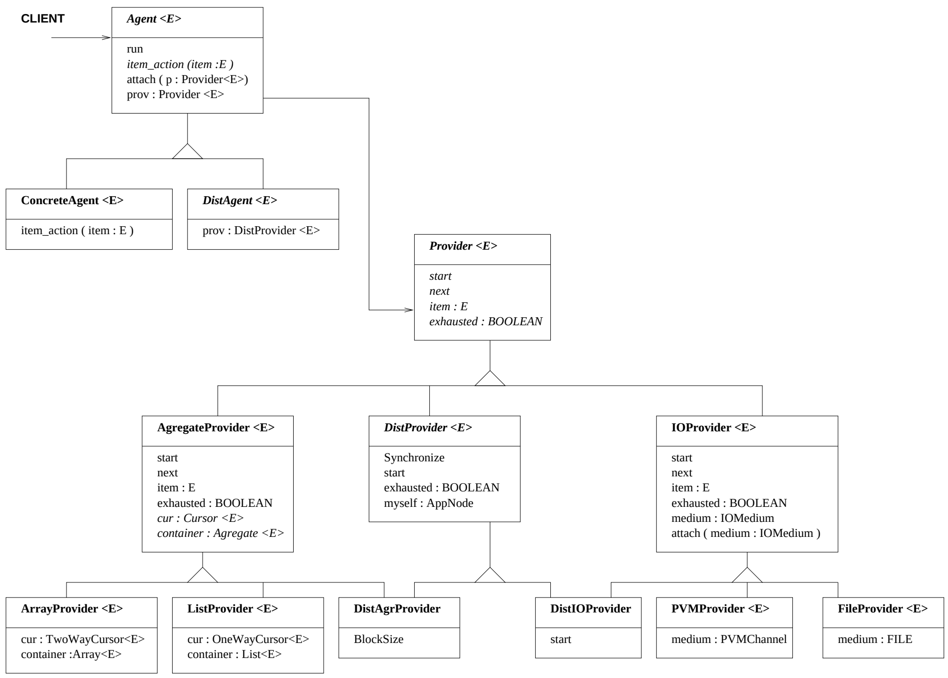

Figure 1: Ubiquitous Agent Pattern Structure

Provider (*deferred*)**:** Defines the common interface for every possible providing process.

AgregateProvider (*concrete*)**:** Implements a polymorphic iteration using a cursor instance. The polymorphic cursor is instantiated by the aggregate this provider is attach to.

ArrayProvider and ListProvider (*concrete*)**:** Implements an iteration over an array or list aggregate. These providers are similar to the aggregate provider but avoid the overhead of polymorphism on the aggregate operation when tuning the application for high performance purposes (all calls can be statically bound and even inlined).

DistProvider (*deferred*)**:** Defines the basic interface for processing elements concurrently. This provider implements a synchronization behavior without specifying an actual traversal policy.

DistAgrProvider (*concrete*)**:** Implements a provider that traverses an aggregate using a distribution of the elements in this aggregate. *BlockSize* is an attribute of this class that may be used with the *myself* attribute to allow the selection of local elements only.

IOProvider (*concrete*)**:** Implements a provider that traverses an I/O medium to reach its elements. The IOMedium class, like the Aggregate class, is an abstract class. Thus, using delegation through polymorphism, an IOProvider is a concrete object based on an abstract class.

DistIOProvider (*concrete*)**:** Implements a provider that traverses an IO medium to reach its elements. The particularity of this provider is that synchronizations inherited from DistProvider are used to ensure a semantically correct execution of the Ubiquitous Agent pattern. Elements are read from the IO Medium which can be either a file or a set of communication channel.

7 Collaborations

The collaboration between an Ubiquitous Agent and a provider object is restricted to the functionality of a generic element provider.

8 Consequences

Using an Ubiquitous Agent independently from the element providers has several benefits:

- The client can define operations without any consideration for the way elements are stored (or not stored) within the application. This allows the separation between conceptual design (concerned with the problem to be solved) from implementation design (concerned with performance related decisions on data structures, sequential and distributed processing, etc.)
- Because the Ubiquitous Agent pattern is inspired from the Iterator one, it inherits all of the benefits of Iterators:
  - Support for several traversal policies
  - Simplify the Aggregate interface
  - Support several concurrent traversal on an Aggregate
  - Support of polymorphic iteration (using Cursors)
- Because the data structure is hidden from the pattern, special aggregate can be built (and are build in the EPEE framework) using distributed or shared memory multi-processors.
- Allow the distribution of control in the provider using polymorphism. The sequential version can use Aggregate provider such as the ArrayProvider and be distributed (under certain dependency constraints) simply by associated a DistAgrProvider in place of the ArrayProvider.

9 Implementation

An implementation of the Ubiquitous Agent pattern has been done in the EPEE project using the Eiffel object oriented programming language. This allowed us to build various distributed applications (e.g. N-body problem simulation, human genome mapping application, etc.) using complex aggregate data structures (e.g. oct-trees).

A first version of the application can be designed, implemented and tested in a sequential environment, before being derivated for a parallel computer. We report in [5] an experience involving this Ubiquitous Agent design pattern on an N-body simulation, that has been efficiently implemented using the shared memory of a Power Challenge Array multi-processors.

References

[1] Grady Booch. *Object Oriented Design with Applications.* Benjamin Cummings, 1991.

[2] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[3] F. Guidec, J.-M. Jézéquel, and J.-L. Pacherie. An object oriented framework for supercomputing. *Journal of Systems and Software*, Special Issue on *Software Engineering for Distributed Computing*, June 1996.

[4] J.-M. Jézéquel. EPEE: an Eiffel environment to program distributed memory parallel computers. *Journal of Object Oriented Programming*, 6(2):48–54, May 1993.

[5] J.-M. Jézéquel and J.-L. Pacherie. Parallel Operators. In P. Cointe, editor, *ECOOP'96 proceedings*, number to be published. Lecture Notes in Computer Science, Springer Verlag, July 1996.