FOCALE: Towards a Grid View of Large-Scale Computation Components

Gaëtan SCOTTO DI APOLLONIA, Christophe GRANSART, and Jean-Marc GEIB

LIFL (Laboratoire d'Informatique Fondamentale de Lille),
USTL (University of Science and Technology of Lille),
F-59655 VILLENEUVE D'ASCQ CEDEX, FRANCE
{scottoda, gransart, geib}@lifl.fr

Abstract. In the model of grid computing, we have several views of the components of the grid. One would prefer a large grid of small data-driven computations, while another one would rather use coarse grain components with great computing power. This paper presents our project, FOCALE, which provides genericity for large components interconnection through Java and CORBA.

Keywords: CORBA and JAVA, distributed computing, objects and components

1 Motivations

Grid computing is today one of the greatest issue in the world of large-scale distributed applications. This concept not only promotes the distributed platform research projects, but it also involve major technical improvements, such as computing centers power, and network throughput and latency. In the United States, large universities and laboratories are already connected by such high-speed backbones, and serve as a testbed for the future of large applications and platforms supporting them. In other countries, whereas it is not as developped, great steps have been taken to attain that goal. In France, the education administration has put up a national network joining the major universities and public laboratories, and this network recently improved its speed with ATM technology. Some researchers also work on an ambitious program making this network attain the Gigabit/s bandwith.

However, in the general case, the main projects around a new item involve creating a new standard, changing everyone's programming method, and developping the applications from scratch. We intend to develop a platform over these networks permitting to any program, already existing or not, to be connected to any other, in a tree of connections that will form our distributed application. Such an application can be whatever the application provider will want, as he can either use already known components, or provide newly-created ones. Applications worth being distributed upon such a large-scale platform include, but are not limited to, all mathematical and applied mathematics computations, database management and datamining, virtual reality and visualization.

Our project is named FOCALE, which stands for Federation of Objects for Computing on a large scALE[1]. As a starting step, FOCALE is being developped in Java, with use of the CORBA standard to permit its future implementation in any[2] language (like C++, for better preformance), and its communication with clients also written in any language. CORBA use is not required from the computing components, because that would mean rewriting some of the world's most used computing libraries, and these libraries are not always written in a CORBA-compliant language, like Fortran.

Let we present some interesting projects, related to ours by the fact that they manage and use either Java components, computing platforms, CORBA technology or world-wide interconnections.

1.1 JAVA-based execution platforms

Ajents [6] is an environment for mobile java applications. With its purposes, it addresses mainly the world of java-written mobile agents. Ajents makes use of JAVA possibilities, and RMI for mobility.

ProActive (formerly known as Java//) [1] is a project which provides "seamless metacomputing" in JAVA. It does not use CORBA, though, but is based on a MOP (Meta-Object Protocol) and issues polymorphism between remote and local objects in a portable JAVA library.

1.2 Components-based environments

NetPebbles [12] is an environment dealing with network components, with the interesting aspect of a script language with which users can create components and interact with the system.

Comet [18] is an architecture based on components, for the construction of distributed applications. It provides means for components to adapt to their environment, as well as acting on it. The model used here is the separation of control from functionalities.

Regis [10] is a system which supports configuration programming, that is, separating the configuration phase from the code. The authors of Regis also made a graphical user interface, the Software Architect's Assistant [13] where users can plug their components one to another, to help the design of the application. They also have written a specifying language, called Darwin [9], that deals with the the high-level organisation of components and the interaction between them.

[1] this has a better meaning in French, but this is close

[2] almost any

1.3 High-performance interconnection

At the hardware level, networks are always increasing their throughput. High-speed national-wide backbones projects already exist in several countries. It is one of the aspect of raising technology to higher levels: it is either for technology itself, or for its users.

The Cobra project [19] proposes a runtime based on CORBA and working on top of a network of multiprocessor workstations, interconnected with SCI [2]. The special involvment here is the use of a high-speed local area network.

1.4 World-wide projects

Legion [8] and Globus [5] are wide-spread projects, that also aim to provide ease of use and fast access to metacomputing to end-users. Legion architecture was designed with computing speed in mind, and multi-language support is, however present, somewhat limited to C/C++ and Fortran/HPF.

Globus and Legion were born before the recent versions of CORBA and their teams are currently working on the integration of CORBA support in them.

Globe [4] is also an object-based framework for developping wide-area distributed applications. Its authors make use of CORBA possibilities to implement worldwide scalable Web documents.

1.5 Incoming CORBA technologies

In the near future, CORBA will be enhanced by some aspects that directly interest us. Some requests for information and proposals have been posted, like *Parallel* CORBA [15] and *Aggregated Computing in* CORBA [14].

The Advanced Computing Laboratory[3] has grouped some of its projects into a response to these RPFs. Ligature [7], one of them, is a database of software components designed to interact with each other through programmer-defined interface.

1.6 Comparing with us...

All of these projects have very interesting functionalities, but were not designed with the same ideas as FOCALE, and as such do not provide the same tools. The following ideas are present in one or more of the described environments, and particularly interest us:

- **Component description language**
  Such a language[4] is important to help end users manipulate FOCALE objects, without entering into the technical part of establishing and maintaining links between their objects.

[3] Los Alamos National Laboratory

[4] whether it is IDL3 or any personally-defined language

– **Corba support**
  "Native" Corba support is usable for the platform itself, and we will see that our platform makes extensive use of it. The console we developped also uses Corba to communicate with Focale servers.
– **Separation of control and functionalities**
  To permit a good developpement of the platform, we separated these aspects, so that developping a control aspect does not hinder the writing of each level's code (see next section for a description of Focale architecture's levels).
– **Genericity of operations**
  The Focale project aims to build applications by interconnecting user-defined computing components. Such components can use Corba to communicate, but some of the best-known implemented computing algorithms don't. As it is, we prefer a platform that can connect and run generic components than focus on one language and/or API and then be stuck with it, forcing willing users to rewrite their code once more. This aspect is the main difficulty of our platform development, but this is also what separates us from most of the previous examples of related work.

By developping these aspects (and others), we do not fit exactly in one of the previous examples, but it is obvious that some areas overlap.

2 Focale Architecture

The construction of our project was made upon a top-down analysis of Grid-computing platform needs. This analysis permitted to separate aspects of the differents execution levels involved. However, in separating levels, we often forget non-functional capabilities, like the persistence of data, or the event management in the different levels. As we are building Focale with Corba, we have the Corba Common Services list in mind not to forget about those capabilities.

Here is a description of the levels, with an brief explanation of how they work, and of the non-functional capabilities associated to them, when applicable.

2.1 The Federation Level

Under this term, we put a group of machines, each with a Focale server running, and interconnected (see figure 1). The functionalities of one server are described in the next paragraph. Users can access this group by any server, although they would prefer to select the nearest one - in terms of either physical distance or network latency. The federation management is not centralized, as it would reduce its global fault-tolerance. Therefore, each server acts as a grid node, with equal power over its local array of components.

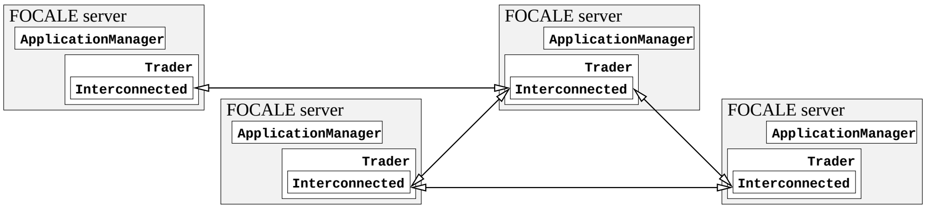

Fig. 1. A Federation consists of interconnected FOCALE servers

2.2 The Server Level

A FOCALE server must be run on a machine for it to be usable with our federation. Although it is not encouraged to do so, some users can propose more than one FOCALE server on one host.

Client interaction

Our server provides an interaction with the users, through OMG-IDL-defined contracts. Among these, we defined the following interfaces, which will be explained in this section:

`Interconnected`, `HasProperties`, `FactorySet`, `FactoryTrader` and `ApplicationManager`.

Server interconnection

A requested capability of our server is to be interconnected with its peers (see figure 1). This can be done by two ways: first, a user can launch a server on a machine, then notify one of the registered servers of its availability; the second way is more automatic, as a starting FOCALE server can browse the CORBA Naming Service, and find there its peers.

The technical part of the connection involves the new server asking another to register, through an OMG-IDL-defined operation `register`. The requested server then invokes a `notifyRegister` on itself and on the asking server, to permit a bi-directional communication.

Server properties

An object implementing our OMG-IDL interface `HasProperties` has a set of properties, that the user can browse and manipulate. Each of these properties is defined by a name and a value. Some properties have a meaning, and a FOCALE server, by checking its environment, sets values for `OS`, `architecture`, `ORB` and `user`, at least.

These default properties are set to permit the management of code execution: if a server runs on a Sun/Solaris, and a code written for a PC/Windows has to be run, it will not be done there. These properties can also be checked to tell if the server's underlying machine works with little-endian or big-endian integer, and so on... Speaking of code leads naturally to the following aspect of a FOCALE server: the `FactorySet`.

Factories set

What is a Factory ? In our project, we used the definition of E. Gamma *et al.* [3] for the *Factory design pattern*. We enter in more details about factories in the section 2.4. For this paragraph, we talk about a Factory set: each FOCALE server implements a way to store many factories, and retrieve them, locally (we will present the "remotely" aspect in the following section). *Persistence* of the `FactorySet` is achieved through individual Factory's persistence, but using a `Cache` mechanism delegated to the server level.

To create a factory in this set, a user has to submit all the `Factory` fields (see section 2.4). Once this is done, the newly-created factory is available for instanciation or local copy by all servers of the federation.

Trading service

For a server to be able to retrieve a Factory that is stored somewhere in the federation, but not locally, we designed a (very) simple trading service. This trader is inspired by the CORBA Trading Service, but not fully compliant, because it is to be used only internally.

How does it work ? First of all, each Factory has a set of properties, for the code name, job description, version, provider, number of uses, etc. and retrieving a Factory is made by using these properties as criteria for a Trader search. For example, you could ask for the most-used matrix inversion component Factory, or the last version of the Factory named "Transpose".

If a corresponding Factory is not present locally, the servers `Interconnected` to this one will be asked for it, in a depth-first tree browsing. Although the servers are interconnected as a graph, cutting the cycles of this graph will form a tree. The cycles are cut when the search process encounters a server it has already browsed.

Applications manager

In the same way as the `FactorySet`, the `ApplicationManager` OMG-IDL interface provides ways to create, obtain and remove Applications to a FOCALE server. The description of one individual Application follows, in the section 2.3.

2.3 The Application Level

In FOCALE, an Application follows the *Factory design pattern*, but at a higher level: an Application is described as a set of interconnected factories, and is instanciated by instanciating the described factories, then connect them (one's output is linked to the other's input, see section 2.6 for description of the *Connectors*) then launch them (see figure 2).

We remark here that instanciating and running a Factory are two different steps. These had to be separated to permit the interconnection between them, so

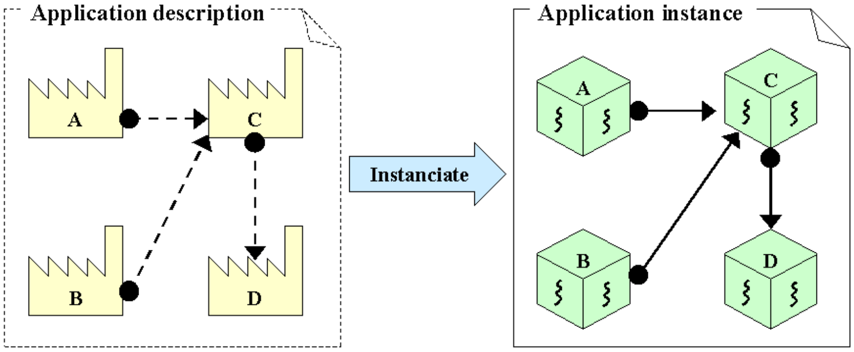

Fig. 2. Instanciating an Application instanciates all its Factories

that the producer does not start its job before having properly been connected to its consumer. Therefore, we will introduce a lower level after the Factory level description: the Instance.

2.4 The Factory Level

Conceptually, a Factory is an object that can create a (possibly) unlimited number of instances of one class. It can destroy them too, although in FOCALE, this could generate some problems: in Java, we are not able to take a snapshot of the stack trace, to restore it afterwards. So, users wanting a factory to destroy its instances must be sure of what they are doing.

In FOCALE, a Factory consists of a `Code` structure and methods to manage it (initializers, instanciate, run...), as well as a `HasProperties` interface implementation (see section 2.2 on Properties). The `Code` structure contains the following items (see figure 3), which can be accessed independantly — as defined in the `Factory` interface — to reduce the amount of unnecessary network traffic.

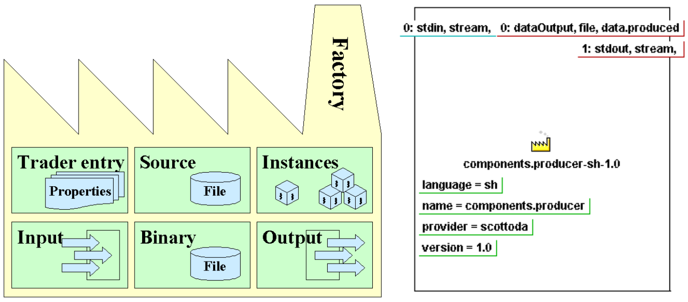

Fig. 3. The factory's content, and its console representation

– `FactoryTrader` **entry**
  This field is a set of properties (see section 2.2, again). Such properties include the component name, its implementation language, version, provider, etc. There is no restriction on the properties, so that each user can add the property he wants, like comments on the code, or its quality of service.

– **information about the code execution**
  In order to be able to launch the given code, the platform must know its execution command in the case of a process-level component, or the name of the main method for a class (see the next section, 2.5, for further details). This field includes also the original code path which is mapped locally in the FOCALE server's cache.
  We also have included a set of trader entries to define the dependencies of the code, that is, a description of the codes to load locally before running this one. In the case of a JAVA class component, this is not mandatory, as our implementation of the `ClassLoader` will do a part of this job.

– **basic knowledge of the component input/output channels**
  An application is not always — in fact, quite rarely — made of a single component, and to be able to connect the components of an Application, the platform has to know where the component is reading from, and where it is writing to. This item contains 2 arrays (one for the input, the other for the output) of the `Resource` structure. This structure contains the following fields: `name`, `type` and `location`. See section 2.6 for details on how the connection is achieved.

– **the code itself**
  When uploading the component into the federation for the first time, its provider has to give its binary form[5], so this is a part of the `Code` structure. However, as we wanted to be able to transmit all the informations about a Factory without requiring this field (which is rather big), we grouped all the Factory fields except this one into a `Descriptor`, which can be accessed directly thanks to the `Factory` interface.

Persistence of a Factory data is done by storing its `Code` in a file. Of course this could lead to problems (with quota-enabled users for example), but we will assume that users proposing their machines as computing resources are not "just" these machines' users.

2.5 The Instance Level

When a Factory is instanciated, it produces an object which implements the `Instance` OMG-IDL interface. This interface defines (among others) methods to connect this instance to others, run the created instance and test if its job is

[5] We can also work from a source code, but this will not be presented here.

finished. As we wanted the platform to be the most generic possible, we have factories handling code that can be either Java classes, or processes (including all other programming languages: C/C++, Fortran, shell scripts, IDLscript programs, etc.). In the case of a Java class, the initialization of the `Instance` finds the given class and its main method, whereas the `launch` function invokes that method. In the case of a process, the initialization just keeps the external command, and the `launch` function executes it.

To connect an instance to another, users have to call the `connectOutput` function on the instance producing data, and `connectInput` on the instance consuming it. Each of these functions return a `Connector` (see next section), that we have to connect to each other, by calling the `connect` method on one, giving the second as argument. As the `Interconnected` aspect of the FOCALE server (see section 2.2 on Interconnection), this will call a notifying method on both connectors. Once an producer instance output if connected to its consumer, all of the data going through the output connector's given resource will be redirected to the consumer's input connector's resource.

If the output connector of an instance is not connected, we will assume that its data will form the result of this component's job. If the user is logged to the federation, he is notified of the termination, and the result is shown. If he logged out after having submitted his job, he will be notified the result when he will log back.

2.6 The Connectors

Conceptually, a `Connector` is like a system pipe, extended to CORBA.

A `Connector` contains a `Resource` structure (see section 2.4), which is used to specify which resource is being used to transmit data. These resource can be the input/output standard streams of the `Instance`, a file or a socket. It is not mandatory for an input connector to map the same resource type as the output connector, as the resource is given for only one of the connectors. Actually, it is possible for a standard output stream to be linked to a file or a socket, for example.

We have plans to provide also URL resource (publishing and fetching data through URLs), and also a Method resource which will be run to fetch data or send it (mostly for specially designed new components, and more in the style of the CORBA Component Model [16]). An output `Connector` consists of a thread watching the given resource, and upon data arrival, sending it to the input connector. When data arrives, the input connector moves it to its given resource, so that it is available to the consumer. These steps are illustrated on figure 4.

It is possible that two components cannot communicate directly, because of the differences of information representation between the produced data and the consumer input format. In this case, the Application provider can put a "translation" component between the consumer and the producer, so that the producer data can be translated into the consumer input format.

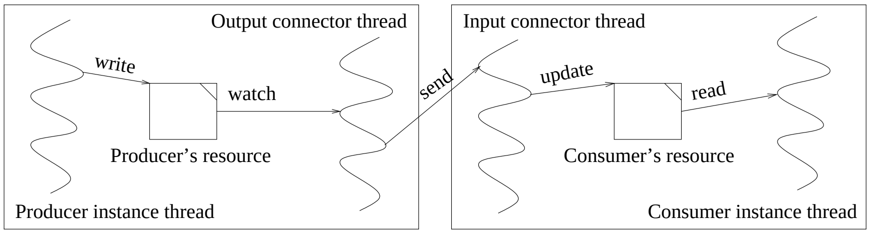

Fig. 4. The steps of the communication between two connected Instances, one producing data and the other consuming it

3 The Console

Using the FOCALE federation can seem complicated, with all these levels and interfaces. Even with CORBASCRIPT [17] [11], it is not easy to feed the servers with new factories, although the manipulating methods are easy to find and invoke. This fact, along with the end-user availability of our platform, made us create a console above FOCALE federation. This console has a graphical user interface, made with Java/Swing, using different frames to display different levels of the federation. With it, you can create/connect/remove servers from the federation, add/move/delete factories from the servers (figures 3 and 5), create/modify/run/destroy applications from the servers (figure 6), and many other functionalities, like catch events from any item[6], or display information on any level of the federation.

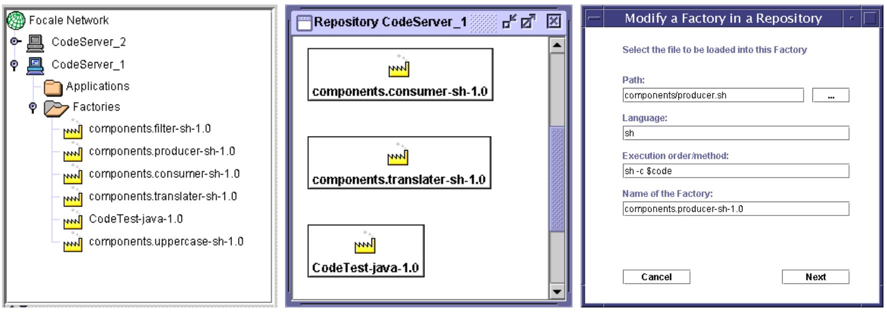

Fig. 5. Console display of a server factories, as a tree and as a window, and the factory installation wizard window

Of course, users can develop their own clients, that can access FOCALE federation through CORBA, by using the OMG-IDL contracts of FOCALE. In this case, the console is not needed, but it can always serve to monitor the client progress.

[6] our implementation of the Observer/Observed design pattern used will not be explained in this document

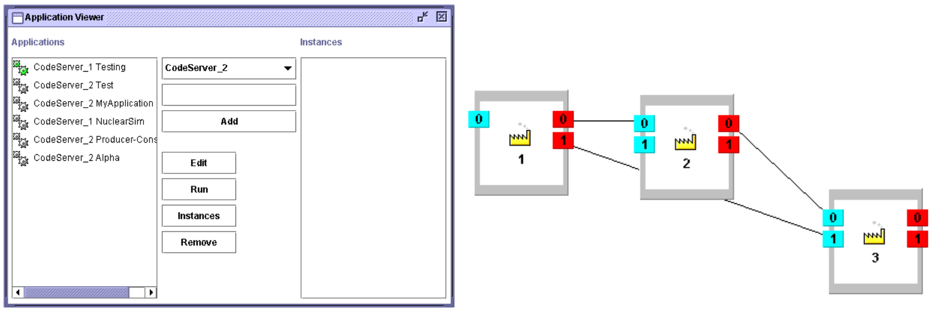

Fig. 6. Console display of the applications list, and one application factories interconnection

4 Conclusion

The platform described in this article, as well as its console is now working, and it is available from `http://corbaweb.lifl.fr`. Generic components can be installed, deployed, and run in applications. It is interesting to note that, although formerly developped only for scientific applications, FOCALE uses enough genericity to permit any form of application to be launched upon it.

The performance of our platform was not discussed here, as it is not the main issue in FOCALE: we provide a simple mean to deploy and interconnect computation components, and when this is done, they run at their own speed. Actually, the performance of the platform itself does not reside in the running speed of these components, but in its capacity to deploy, find and connect them.

Future works

Globally, the platform is not very fault-tolerant (although if one server is missing, others can fetch factories from elsewhere), due to the evident fact that computing nodes spread over the Internet are not very reliable. One of the main next step of FOCALE improvement will then be improving its fault-tolerance and quality of service.

It has also been noted that the performance of the platform was somewhat limited to loading and connecting components, but if a server becomes too loaded, there is no way, yet, to reduce it. To deploy components, we will use theoretical issues such as optimisation and graph theory, that can provide means to speed up the positioning of components on the federation and the search of those components afterwards. For the components' execution, we intend to include a load balancing policy to increase the performance of this phase.

To the end-user point of view, it would be interesting to have a "preview" of the deployment and execution of its application. Therefore, we intend to include a cost prevision module, which will present a rough measure of time for the deployment and execution phases. Such a module will obviously display its information for special use of the platform and the components, as it is not known beforehand if other users will use it, and if servers will crash during the process.

References

1. Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. In Geoffrey C. Fox, editor, *Concurrency Practice and Experience*, volume 10 (11-13), pages 1043–1061. Wiley and Sons, Ltd, Sep/Nov 1998.
2. Paul Sweazey *et al.* SCI - Scalable Coherent Interface, P1596/D2.00. Technical report, 1992.
3. E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Westley Professional Computing, USA, 1995.
4. Globe web site.
   `http://www.cs.vu.nl/globe`.
5. Globus web site (papers available).
   `http://www.globus.org`.
6. Matthew Izatt, Patrick Chan, and Tim Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In ACM 1999, editor, *Java Grande Conference*, pages 15–25, June 1999.
7. Katarzyna Keahey, Peter Beckman, and James Ahrens. Ligature: Component Architecture for High-Performance Applications.
8. Legion web site (papers available).
   `http://legion.virginia.edu`.
9. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conference*, September 1995.
10. J. Magee, N.Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5):304–312, September 1994.
11. P. Merle, C. Gransart, and J.-M. Geib. Using and Implementing CORBA Objects with CorbaScript. *Object-Oriented Parallel and Distributed Programming*, 2000. Ed. Hermes.
12. Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, and Murthy Devarakonda. Programming Network Components Using NetPebbles: An Early Report. In *USENIX Conference on Object Oriented Technologies*, April 1998.
13. K. Ng, J. Kramer, and J. Magee. A case tool for software architecture design.
14. Object Management Group. Supporting Aggregated Computing in CORBA. OMG TC Document 1999-01-04, Object Management Group, First Needham Place, 250 First Avenue, Suite 201, Needham, MA 02194, U.S.A., 1999.
15. Object Management Group. Parallel Application Support for CORBA. OMG TC Document 2000-02-06, Object Management Group, First Needham Place, 250 First Avenue, Suite 201, Needham, MA 02194, U.S.A., 2000.
16. OMG. *CORBA Components: Joint Revised Submission*. Object Management Group, August 1999. OMG TC Document orbos/99-07-{01..03,05} orbos/99-08{05..07,12,13}.
17. OOC and LIFL. *CORBA Scripting - Joint Revised Submission*. Object Management Group, August 1999.
18. Frédéric Peschanski. COMET, reflective architecture based on components, for the construction of parallel and distributed applications.
19. Thierry Priol and Christophe René. Cobra: a CORBA-compliant Programming Environment for High-Performance Computing. In *Europar'98, Southampton, UK*, pages 1114–1133, September 1998.