THÈSE DE DOCTORAT

École Doctorale
« Sciences et Technologies de l'Information et de la Communication »
de Nice - Sophia Antipolis

Discipline Informatique

UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS
FACULTÉ DES SCIENCES

COMPONENTS FOR GRID COMPUTING

par

Matthieu MOREL

au sein de l'équipe OASIS,
équipe commune de l'I.N.R.I.A. Sophia Antipolis, du C.N.R.S. et du laboratoire I3S
soutenue le 27 novembre 2006

Jury:

Rapporteurs	Dennis GANNON	Professeur, Indiana University
	Sergei GORLATCH	Professeur, Westfälische Wilhelms-Universität
	Thierry PRIOL	Directeur de Recherche, INRIA
Examinateurs	Thierry COUPAYE	Expert Recherche, France Télécom
	Philippe LAHIRE	Professeur, UNSA
Directeur de thèse	Denis CAROMEL	Professeur, UNSA

To my family,
– Matthieu

Acknowledgements

The course of a PhD thesis is a great opportunity not only to work on exciting topics and challenging problems, but also to meet and collaborate with extraordinary people. For these reasons I deeply thank Isabelle Attali who created, through admirable determination, the best conditions for my PhD thesis.

I thank Denis Caromel, my supervisor, for his trust and support in my work, and for his spirit of innovation. He managed to establish a comprehensive and consistent framework that federates the research efforts of our team, and he put constant incentive in giving a significant impact to my work.

Collaborating with Françoise Baude and Ludovic Henrio was delightful, both from a scientific and human point of view, and I express my gratitude to them. They were very helpful and always available.

My greetings also go to Dennis Gannon, Sergei Gorlatch and Thierry Priol who honored me by accepting to review my thesis. I am also very grateful to the other members of the jury, Thierry Coupaye and Philippe Lahire.

I respectfully acknowledge the work of the authors of the Fractal model: they made a major contribution to the field of middleware technologies, and brought a great platform to the scientific community.

Many thanks to Mario Leyton for his thorough review of this document and to Christian Delbé for his enlightening comments.

Thanks to Nikos Parlavantzas for his contribution to the design and to the reporting of the methodology for the Jem3D experiment, and thanks to the first users of the component framework for their valuable feedback.

I would also like to thank all the members and collaborators of the Oasis team for very insightful scientific (and not only scientific) discussions and contributions. In random order: Alex, Laurent, Fabrice, Santosh, Antonio, Marcelita, Paul, Guillaume, Géraldine, Romain, Clément, Arnaud, Virginie, Eric, Tomas, Igor, Vladimir, Javier, Yu, Luis, Maciej... among others.
Working in such a dynamic, emulating and diversely rich team was a pleasure. I believe it is a key factor for creating the conditions of successful achievements.

Of course, I cannot but express my best greetings to my parents Alain and Jacqueline and to my sister Elise for their help and support.

Last but not least, I would also like to thank the members of the INRIA squash team, particularly Francis, for their great sports spirit and for very enjoyable extra-thesis moments.

Contents

List of Figures

List of Tables

Glossary

Active object A distributable object with its own thread and its graph of non-shared passive objects. Communications with active objects may be asynchronous through a transparent future mechanism.

ADL Stands for Architecture Description Language: a file defining a component system

Annotation Metadata added on a declaration in order to add semantic information to an element.

Autonomic computing A software paradigm that defines self-managed components able to monitor their environment and adapt to it by automatically optimizing and reconfiguring themselves

Broadcast Mode of distribution of parameters in a multicast communication in which all receivers are sent the same data.

Component-based programming Programming model in which software units are units of deployments with contractually defined interfaces.

Composite binding An indirect connection between components through dedicated binding components.

Controller In the Fractal model it is a constituent of a component and it handles non functional properties.

Core Grid Middleware Software layer offering management and utility services.

Deployment descriptor A file that defines the mapping between a virtual infrastructure based on virtual nodes and a targeted physical infrastructure.

Fractal A modular and extensible component model that can be used with various programming languages for the design implementation deployment and reconfiguration of software systems.

Future A placeholder for the result of an invocationİt is given as a transparent proxy to the actual result and automatically updatedİt enables transparent asynchronous invocations.

Gathercast Communication from n entities to 1 entity. A gathercast interface transforms a list of invocations into a single invocation.

Grid A shared computing infrastructure of hardware software and knowledge resources that allows the coordinated resource sharing and problem solving in dynamic multi-institutional virtual organizations to enable sophisticated international scientific and business-oriented collaborations.

Grid Application Layer Grid software layer containing the Grid-enabled applications.

Grid Fabric Accessible distributed and federated resources of a Grid infrastructure.

Grid Programming Layer Grid software layer offering high-level abstraction for programming and deploying Grid applications.

Intercession Ability to modify the characteristics of a software entity at runtime.

Interface cardinality In the original Fractal specification indicates how many interfaces of type T a given component may have. This was extended in our proposal for designating interfaces with multiple bindings (multicast gathercast).

Introspection Ability to inspect the characteristics of a software entity at runtime.

Multicast Communication from 1 entity to n entities. A multicast interface transforms a single invocation into a list of invocations.

MxN problem Refers to the problem of communicating and exchanging data between parallel programs from a parallel program that contains M processes to another parallel program that contains N processes.

Network-enabled server Programming model where tasks are delegated to workers through a dedicated server.

Parallel component A component that exhibits at least one multicast server interface.

Reflection Ability to introspect and intercede on a software unit at runtime.

Scalability The ability to handle large number of entities.

Scatter Mode of distribution of parameters in a multicast communication in which receivers may be sent different data.

Scavenging Grid A distributed computational infrastructure that uses idle processor time from volunteering machines.

Service Oriented Architecture Programming model that emphasizes decoupled collaboration between software units.

Skeleton High-level and parameterizable algorithmic pattern.

SPMD Stands for Single Program Multiple Data: a programming model for parallel computing in which each task executes the same program but works on different data.

Tensioning Optimization mechanism that remembers the shortest way between two software entities and establishes a direct connection between them.

Virtual node An abstraction of the physical infrastructure used in application code in order to separate the design from the infrastructure.

Part I

Thesis

Chapter 1

Introduction

Computer chips performance follows Moore's law, storage capabilities are also increasing exponentially, access to computers and electronic equipments has generalized in industrialized countries, and the Internet and wireless technologies provide worldwide and ubiquitous interconnection. As a result, we are surrounded with increasing computational capabilities. They present new opportunities, for scientists to solve new categories of problems, for industrials to develop new business models, and for academics to investigate ways to use these resources.

Taking advantage of these resources implies federating them: this is a difficult challenge. The term *Grid computing* was coined in the 90's to designate this challenge, and Grid computing was popularized with the book "The Grid: Blueprint for a New Computing Infrastructure" [FOS 98]. The concept of Grid has since matured and may now be defined as follows:

> *A production Grid is a shared computing infrastructure of hardware, software, and knowledge resources that allows the coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations to enable sophisticated international scientific and business-oriented collaborations* [LAS 05].

1.1 Problematics

Grid computing has specific requirements due to its inherently largely distributed and heterogeneous nature. Research efforts first focused on the access to physical resources by providing tools for the search, reservation and allocation of resources, and for the construction of virtual organizations. Accessing the Grid infrastructure is a first necessary step for taking advantage of the resources of the Grid, and the second step is to offer adequate development models and environments (frameworks). A new research area therefore emerged, which focused on *programming models and tools* for efficiently programming applications which could be deployed on Grids. As mentioned in [FOS 98],

> *Grid environments will require a rethinking of existing programming models and, most likely, new thinking about novel models more suitable for specific characteristics of Grid applications and environments.*

The requirements of Grid computing that must be handled by programming environments may be listed as follows:

- *Distribution* : resources can be physically distant from each other, resulting in significant and sometimes unpredictable network latency and bandwidth reduction.
- *Deployment* : the deployment of an application on a Grid is a complex task in terms of configuration and connection to remote resources. The target deployment sites may be specified beforehand, or automatically discovered by browsing available resources and by selecting those matching deployment constraints.
- *Multiple administrative domains* : the resources can be spread around many different networks, each with their own management and security policies.
- *Scalability*: Grid applications use a large number of resources, and the framework must be able to accommodate large number of entities, notably by handling latency and by offering parallelism capabilities.
- *Heterogeneity* : unlike cluster computing, where resources are homogeneous, Grids gather resources from multiple hardware vendors, running on heterogeneous operating systems, and relying on different network protocols.
- *Interoperability and legacy software* : in order to reuse already existing and optimized software, legacy software should be wrapped, enabling further integration in broader systems, and interoperability with tier applications must be possible.
- *High performance* : programming frameworks should be designed for efficiency, notably by offering parallel programming facilities, because one of the aim of Grids is to solve computation-hungry problems.
- *Complexity* : Grid applications can be complex software because they combine the complexity of highly specialized pieces of software with integration, configuration and interoperability issues.
- *Dynamicity* : the application may evolve at runtime (structurally and functionally), motivated by environmental changes (for ensuring optimized performance, or in case of failures), or by the addition or removal of resources while the application is running (a Grid is intrinsically dynamic).

1.2 Objectives and Contributions

This work belongs to the research area that focuses on programming models and tools, and our main objective is to *define a programming model and a framework that address the requirements of Grid computing*.

We consider that some of the requirements must be handled by the underlying Grid middleware, such as distribution, multiple administrative domains and interoperability, and that the other requirements must be handled by the programming model. In the programming model we propose, we aim at simplifying the design of Grid applications, and at providing high-level abstractions for the design of interactions between multiple distributed entities.

Our approach relies on *a hierarchical component model based on active objects*, augmented with *collective communications*.

The main contributions of this thesis are:

- an analysis of the adequacy of existing programming models, and particularly component models, with respect to the requirements of Grid computing,

- a proposal for a component model that aims at fulfilling these requirements, through hierarchical composition, and a component deployment model suitable for Grid computing,
- a specification of collective interfaces as a means to design multiway interactions between distributed components, parallelism, and synchronization between multiple components,
- a component framework that implements the proposed component model and the specification of collective interfaces.

1.3 Overview

This document is organized as follows:

- In Chapter 2 we position our work in the context of the Grid software architecture. We provide an overview of existing programming models for Grid computing in order to justify our approach, and we evaluate means for collective communications. We consequently expose how Grid computing requirements are addressed in existing models and frameworks; this allows us to point out what must be enhanced. Finally, we present the active object model and the ProActive middleware that we based our work on.
- In Chapter 3, we propose a component model for Grid computing that grounds up on the Fractal component model.
- In Chapter 4 we augment the proposed component model with a specification of collective interfaces, as a means to tackle multiway interactions at the level of component interfaces. We show how collective interfaces can be advantageously applied to coupled applications.
- Chapter 5 describes our component framework which implements the component model specified in chapters 3 and 4.
- Chapter 6 reports some experiments we realized using our component framework: we demonstrate its suitability for Grid computing, its scalability, and we show how it can be used for SPMD applications. We also present a methodology for refactoring object-based applications into component-based Grid applications, and we highlight some problematics related to coupled applications in a Grid environment.
- In Chapter 7 we give an overview of our current work and of the enhancements we foresee, and we outline the potential of our model in academic and industrial contexts.
- Chapter 8 concludes and summarizes the major achievements of this thesis.

Chapter 2

Positioning and State of the Art

In this chapter, we justify our choice of a hierarchical component model based on active objects and augmented with collective communications, by evaluating existing programming models for the Grid, by evaluating different approaches to provide collective communications, and by pointing out what requirements are not addressed by existing models and frameworks.

This chapter is organized as follows. We begin by positioning our work with respect to, first, the Grid software architecture, and second, the existing Grid programming models. We thereby justify our choice of a component-based approach. We then present an overview of component-based programming and of common frameworks, and we evaluate component frameworks specifically targeting Grid computing. We also consider the existing strategies for defining collective communications. Based on our evaluations, we discuss the suitability of the existing component models in the context of Grid computing, by considering their general properties and how they fulfill Grid computing requirements. Finally, we describe the active object model and the library we based our work on.

2.1 Positioning

Grid computing aims at providing transparent access to computing power and data storage from many heterogeneous resources in different geographical locations - this is also called virtualization of resources. Taking advantage of these resources involves both an environment providing the virtualization, and a programming model and deployment framework so that applications may be designed, deployed, and executed.

We aim at defining a high-level programming model along with a deployment framework for Grid applications. In this section, we position our work in the context of the Grid layered infrastructure and of the Grid programming models, in order to fulfill the requirements expressed in chapter 1.

2.1.1 Positioning with respect to general Grid architecture

We start by describing our target audience, then we position our contribution with respect to the Grid software layers.

From a user point of view, Grid application developers may be classified in three groups, as proposed in [GAN 02]: the first group consists of *end users* who build packaged Grid applications by using simple graphical or Web interfaces; the second group consists of *programmers* that know how to build Grid applications by composing them from existing application "components"; the third group consists of *researchers and experts* that build the individual components. In this work we essentially address the second group of users, by providing them *a programming model* and *a deployment and execution environment*.

From a software point of view, the development and execution of Grid applications involves three concepts which have to be integrated : *virtual organizations*, *programming models* and *deployment and execution environments*.

Virtual organizations are defined in the foundation paper [FOS 01] as:

> *individuals and/or institutions who share access, in a highly controlled manner with clearly defined policies, to computers, software, data and other resources required for solving computational problems in industry, science and engineering.*

Virtual organizations are set up following concertations and agreements between involved partners. Such organizations are becoming popular within the academic community, and they take advantage of large infrastructures of federated resources such as Grid'5000 [CAP 05] and NorduGrid [EER 03] in Europe, TeraGrid in the U.S. [TER], Naregi [NAR] and ChinaGrid [JIN 04] in Asia, etc.. Global interactions between these infrastructures may also be created punctually, resulting in even larger virtual organizations, such as during the Grid PlugTests events [GPT05, GPT].

Programming models provide standards and a basis for developing Grid applications; the definition of standards is a complex and lengthy process, which has to take into account existing standards used in Internet technologies.

The execution environments allow access to Grid resources, and usually require complex installation and configuration phases.

The concepts of virtual organizations, programming models and deployment and execution environments may be gathered in a layered view of Grid software, depicted in figure 2.1.

- At the bottom, lies the Grid Fabric, which consists of all the accessible distributed resources. These resources physically consist of CPUs, databases, sensors and specialized scientific instruments, which are federated through a virtual organization. These physical resources are accessed from the software stack thanks to operating systems (and virtual machines), network connection protocols (rsh, ssh, etc.) and clusters schedulers (PBS, LSF, OAR, etc.).
  The resources of this layer are federated through virtual organizations.
- Above, layer 2, is the Grid middleware infrastructure (sometimes referred to as the Core Grid Middleware), which offers core services such as remote process management and supervision, information registration and discovery, security and resource reservation. Various frameworks are dedicated to these aspects. Some of them are global frameworks providing most of these services, such as the Globus toolkit [FOS 05a] which includes software services and libraries for resource monitoring, discovery, and management, plus security and file management. Nimrod [ABR 95] (a specialized parametric modeling system), Ninf [SEK 96] (a programming framework with remote procedure calls) and Condor [THA 05] (a batch system) take advantage of the Globus toolkit to extend their capabilities towards Grid computing (appending the -G suffix to their name). Unicore [UNI] and Gridlab [ALL 03] services are other examples of global frameworks providing access to

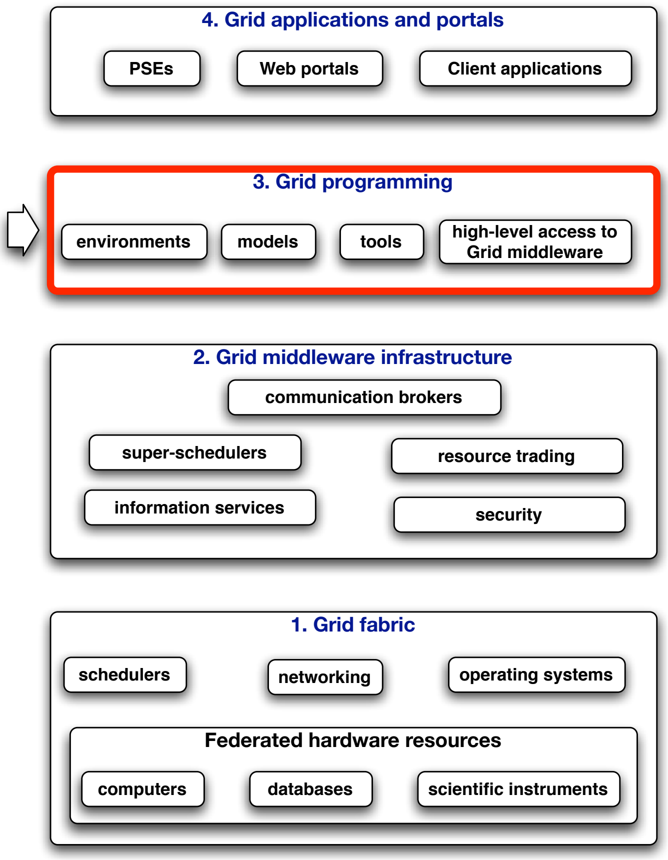


Figure 2.1: *Positioning of this thesis within the Grid layered architecture*

federated Grid resources, with services including resource management, scheduling and security. Other frameworks in the Core Grid Middleware layer are more specialized: JXTA [JXT] for example is a peer-to-peer framework which may be integrated within Grids for managing large amounts of data, such as in JuxMem [ANT 05]. GridBus [BUY 04] is a Grid middleware designed to optimize performance/cost ratios, by aggregating the most suitable services and resources for a given task.
The deployment and execution environments for Grid entities are provided by the Grid middleware layer.

- Layer 3 is the Grid Programming Layer, which includes programming models and tools.
  This layer contains high-level programming abstractions facilitating interaction between application and middleware, such as the GGF's initiative called SAGA [GOO , GGF04] which is inspired by the Grid Application Toolkit (GAT) [ALL 05], and Globus-COG [LAS 01] which enhances the capabilities of the Globus Toolkit by providing workflows, control flows and task management at a high level of abstraction. As reported by some users [BLO 05] however, some of these abstractions still leave the programmer with the burden to deal with explicit brokerage issues, sometimes forcing to use literal host names in the applicative code.

- The top layer is the Grid Application Layer, which contains applications developed using the Grid programming layer as well as web portals, in which users control applications and feed data through web applications. Applications may collaborate in various levels of coupling, from service based interactions to optimized direct communications between internal entities.

The contributions of this thesis belong to the Grid programming layer. Existing Grid programming models are considered in more detail in the following section.

2.1.2 Positioning with respect to existing programming models for the Grid

We describe here the main programming models applicable to Grid computing. A number of articles ([LAF 02, PAR 05, FOX 03, LEE 01] present overview of Grid programming models and environments, highlighting usability as a fundamental requirement. In this section our objective is to justify why we consider component-based programming as the most suitable paradigm.

2.1.2.1 Message passing

Message-passing programming models are popular within the scientific community, as they represent an efficient mean to solve computational problems on dedicated clusters, notably using the SPMD programming style. Message passing frameworks such as MPI or PVM [GEI 94] provide an optimized communications layer. Moreover, numerous algorithms are available for further optimizing the communications and distribution of tasks between involved distributed processes.

Message-passing is one of the most basic ways to achieve process communication in the absence of shared memory, but it is a *low-level* programming style, which on one hand provides efficiency, latency management, modularity and global operations, but on the other hand fails to provide the abstractions necessary for Grid computing, such as support for heterogeneity, interoperability, resource management, adaptivity and dynamicity.

Nevertheless, in regard of the advantages given by latency management, modularity and global operations, some frameworks have been implemented in order to bring the message-based programming paradigm to Grid computing. They usually rely on existing Grid middleware, such as Globus. This is the case for instance of MPICH-G2 [KAR 03], which uses Globus for authentication, authorization, executable staging, process creation, process monitoring, process control, communication, redirection of standard input and output and remote file access. PACX-MPI [KEL 03] is another implementation of MPI geared towards Grid computing, which contains a wide range of optimizations for Grid environments, notably for efficient collective communications. MagPie [KIE 99] also focuses on the efficiency of collective communications: it can optimize a given MPI implementation by replacing the collective communication calls with optimized routines.

In conclusion, although not a high-level programming model, message-passing may be used for Grid programming by relying on the services of other Grid middleware, however, direct control over low-level communications is attractive only for certain kinds of applications.

2.1.2.2 Distributed objects

The object-oriented paradigm is a widely used programming approach, thanks to its high level and structured programming concepts. Distributed communications between objects are easily performed through remote method invocations, either through a standardized extension of the language (such as Java RMI) or through a tier middleware layer (CORBA). The portability of some object-oriented languages such as Java facilitated the development of distributed objects frameworks, and some of them are generalized to address the specificities of Grid computing. For example, JavaSymphony [JUG 04] provides explicit control over locality, parallelism, synchronization and load balancing of distributed objects. *Active objects* are medium-grained distributed objects with their own configurable activity, and they exhibit determinism properties. The ProActive library is an implementation of the active object model; it is described in more detail in a following section, as the implementation work in this thesis is based on this library.

Satin [NIE 05] extends the Java language for providing parallel execution of method invocations, which are reified and can be distributed to the best suited resources. It targets divide-and-conquer programs by offering dedicated constructs (spawn and sync), and automatically load-balances the invocations. Satin uses an optimized communication layer called Ibis Portability Layer [NIE 02]. Both Satin and Ibis Portability Layer belong to the Ibis project [IBI].

Unfortunately, object-oriented approaches impose explicit dependencies between applicative objects, and generally lack modularity as well as external description and configuration, and packaging capabilities.

2.1.2.3 Skeletons

Skeletons are high-level and parameterizable algorithmic patterns, introduced in [COL 89]. Complex applications may be modeled and built by composing basic skeletons such as farm, pipe, map etc., hence keeping a highly structured design. Skeletons are a way to program Grid applications by providing abstraction for parallelism (through parameterizable parallel modules as skeletons), and by providing a higher abstraction level on the program structure, as suggested in [ALT 02]. Programmers use parallelized algorithms built with skeletons and may customize the skeletons with application-specific parameters, allowing optimized implementations.

Several frameworks provide skeleton programming facilities for Grid computing. In ASSIST (A Software development System based upon Integrated Skeleton Technology) [ALD 06] programs are defined as generic graphs: nodes are parallel or sequential modules and arcs represent typed asynchronous streams of data/objects. The classical skeleton programming model is extended to provide flexibility and to express new forms of parallelism. Parallel modules are expressed as so-called *parmod* skeletons. ASSIST provides a high-level language with a compiler, as well as runtime support dynamically enforcing quality of service requirements by means of self-configuration and self-optimization. ASSIST provides interoperability with CORBA and plans interoperability with Globus for accessing Grid services. Another skeletons framework also provides interoperability with Grid services through Globus: HOC-SA [DUN 04], which uses the terminology *higher-order components* (HOC) to emphasize the possibility of composing skeletons.

We observe that current work around skeletons for Grid computing focus on structured and optimized distributed and parallel programming in order to achieve high performance. Grid computing offers a wider diversity of programming challenges and the latest developments in Grid skeletons programming tend to converge with another programming model: component-based programming.

2.1.2.4 Remote Procedure Calls for Grids

The concept of Remote Procedure Call (RPC) has been widely used in distributed computing for many years. It provides an elegant and simple abstraction that allows distributed software entities to communicate with well-defined semantics. Bringing RPC to Grid computing implies adding a number of features for scalability and abstracting from the underlying infrastructure.

Grid-based Remote Procedure Call may be seen as standard RPC augmented with asynchronous coarse-grained parallel tasking. GridRPC [SEY 02] is a proposal fostered by the GGF research group on programming models, and therefore follows a standardization process with the intention of becoming a recognized standard.

In GridRPC, remote calls are identified by session IDs in order to add functionalities such as canceling, waiting for an asynchronous call to complete, querying about past calls. A variety of features are also added in order to hide the dynamicity, security and instability of the Grid from the programmers, but a driving idea is to focus on a simple and lightweight implementation of RPC functionality which would meet the needs of scientific computing. It differs in this sense with standard distributed object-based RPC technology (CORBA or Java RMI) which have broader goals but at the cost of IDL complexity, protocol performance[1] and software footprints.

GridRPC is commonly realized with a delegation model: clients submit invocations to a pool of servers through intermediate agents. These agents gather informations on the servers and therefore handle the scheduling and load-balancing of tasks/invocation in the most suitable manner. GridRPC with delegation is usually referred to as the *network-enabled server* (NES) paradigm. Some examples of frameworks that enable the NES paradigm are Ninf-G [TAN 03], Netsolve [SEY 05] (oriented towards scientific solvers) and DIET [DES 04] (which proposes an innovative distributed scheduling approach for a better scalability).

Scavenging Grids such as BOINC [AND 04] represent a popular Grid infrastructure where the programming paradigm is also similar to GridRPC: the system is centralized and tasks are distributed following a master-slaves model. Such infrastructures are however dedicated to a single category of problems, namely embarrassingly parallel problems, in which the computation can be split into independent tasks with a low data/compute ratio.

Unfortunately, GridRPC is restricted to client-server interactions: it may be used to distribute tasks to servers, but falls short when more complex interactions are required, for example in the case of tightly coupled applications.

2.1.2.5 Service Oriented Architectures and workflows

In this paragraph, we consider the use of services for applicative design. Service based designs provide interoperability, enable on-demand access and loose-coupling, and are a way to achieve scalability. They are also gaining popularity in the industry because they are seen as a way to simplify design, enable code reuse, and facilitate integration of tier products and collaboration with tier companies, as pointed out in [VOG06]. Services are usually composed thanks to workflow languages and workflow engines.

We distinguish the use of a service-based approach for applicative design from the use of services as a mean to access the Grid infrastructure. Services are advocated for Grid middleware since the foundation paper "The Physiology of the Grid" [FOS 02]

[1]CORBA can prove highly efficient on this aspect

which defines the Open Grid Services Architecture (OGSA), and proposes a standardization effort towards a common services-based architectural vision of the Grid. Indeed, as pointed out in [LAS 05]:

> *A job submission, a file transfer, and an information query should not be treated differently and [...] they present a task, or job, to be executed by a service. The task is invoked in all cases through a query to a service.*

OGSA first spawned the Open Grid Services Infrastructure (OGSI) as an implementation standard, which extended web services by adding state, persistency, notification and lifecycle management, thereby defining *Grid services*. This initiative unfortunately failed to converge with existing web services standards. A second initiative is the Web Services Resource Framework (WS-RF) [WSR04], which is more tightly coupled with standard web services and allows integration of web services directly into the Grid fabric; it is implemented in the latest version of the Globus toolkit (GT4).

Coming back to the applicative design on Grids using a service-oriented approach, one should note that Grids are now not merely seen as frameworks for high-performance simulations in academic environments. In the industry (as shown for instance in [SRI 05]), Grid technologies tend to be recognized as *foundations* for flexible management and use of global IT resources as commodity resources; moreover, new kinds of applications can now be envisaged: applications which would build global interactions between various kinds of functional modules and services. In the scientific community, the collaborative opportunities provided by a standardization of interoperable services will be fostered by Grids, which enable a separation of concerns between discipline-specific content and domain-independent software and hardware infrastructure [FOS 05b].

The orchestration of services into workflows requires an adequate workflow language, and a workflow engine to coordinate the participating entities at runtime. Many workflow languages are available for Grid computing, as shown by [YU 05], and a de facto standard has not yet emerged even though some industrially established workflow standards such as BPEL seem extensible enough to suit the needs of Grid computing, as explained in [SLO 06]. There are three ways of creating concrete workflows. The first one is to handwrite them, a usually tedious task. The second one is to use a graphical tool, such as Problem Solving Environments (PSE) graphical interfaces [TAY 05, ALL 02, MAY 03]. For scientists or other Grid applications designers who are not expert programmers, assembling coarse grained applications by workflow composition (either automatically from a program or by graphical composition) is simpler and better suited than lower-level coding and assembly. The third way is to automatically infer the workflow by analyzing a program or a script, which allows an automatic and potentially optimal parallelization of the tasks - an interesting approach is Grid superscalar [BAD 03], inspired by the superscalar design of parallel processors and targeting applications composed of coarse grained entities as these of the Grid NAS benchmarks [FRU 01a].

We note however that Grid services are not suitable for tightly synchronized applications[2], because of the communication overhead of XML/SOAP mechanisms[3]. Furthermore, the deployment of services is not properly specified (services must be up and running, but they somehow need to be placed on their host environment).

[2]Tightly synchronized applications notably include climate, physics and molecular models employing explicit iterative methods.

[3]We note that some implementations of SOAP may provide optimizations for high performance computing [CHI 02]

Finally, we also note that services in service-oriented architectures are coarse grained entities, but that they usually depend on an object-based or component-based framework for their implementation.

2.1.2.6 Component-based programming

Component-based programming is another programming model used for Grid computing. There is no standard definition of what a software component is, although several definitions are well accepted. In the context of this work, we adopt the following definition, proposed after the first edition of the Workshop on Component Oriented Programming [SZY 96]:

Definition 2.1.1 *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

We argue that component-based programming encompasses the formerly described programming models and can be used with new programming paradigms. It either already provides most of the features of formerly described programming models, or can provide them by means of encapsulation:

- Components are a higher level of abstraction than objects, thus they inherit the properties of distributed objects programming models.
- Skeleton patterns are commonly encapsulated in components.
- Communications among components may use message-passing and remote procedure calls, may be synchronous or asynchronous, and may use any optimized communication layer; component-based programming not only allows message-passing and RPC models to be used for communications and task management, but it is also possible to implement these paradigms using component-based programming, as in the DREAM project [LEC 04].
- Components may be exported as services, moreover, components suit both tightly and loosely coupled systems, whereas services target loosely coupled systems. The recent Service Component Architecture specification [SCA] may appear as a convergence between services and components, however it specifically targets loosely-coupled systems which may be decomposed as modules with input and output interfaces.
- Components can handle various levels of granularity, from fine-grained to coarse-grained.
- Encapsulation provides access to high-performance legacy codes.

For these reasons, we decided to follow an approach for the design and execution of Grid applications based on the software component paradigm. In section 2.2 we describe more thoroughly this paradigm and present major industrial and academic component models, and in section 2.3 we present component models geared at Grid computing.

2.2 Component-based programming models and frameworks

In this section we consider the benefits of a component-based approach; then we study some of the most common component models, both industrial and academic. They are not oriented specifically towards Grid computing, but they propose and implement core concepts of component-based programming, and some of these concepts may be advantageous for Grid computing. A summary of component models and component-based programming concepts is proposed in section 2.5.

2.2.1 Component-based programming

Component based programming takes the idea of using construction bricks and elements for building manufactured products, and applies it to software. Historically, the idea of envisaging software development as the assembly of components dates back the a 1968 NATO's meeting, where McIllroy introduced the notion of software components. It is not until the 90's that component based programming effectively broke out in the industry, notably with Microsoft's Component Object Model and .NET, and Sun's Enterprise JavaBeans.

Component-based programming is advocated for three main aspects:

- *Encapsulation*: components are seen as black-boxes, which define services offered and services required through external interfaces; they enforce a strict separation between declaration and implementation, and a strict separation between functional and non-functional concerns.
- *Composition*: components are composable, which facilitates the design of complex systems by simply assembling functional software units, and hierarchical models offer abstractions of composed sub-systems.
- *Description*: assemblies of components are usually defined in *Architectural Description Languages*, which give a higher level of abstraction than raw code, and standardized ADLs means *tools* can be used for the design of component systems, and for verifying the correctness of the assembly. The description of components and component systems can also include packaging and deployment information, to be used during the deployment phase.

The software development process benefits from this approach, which better separates roles within the development team - design, development, configuration, deployment - and increases productivity as the execution environments handle most of the non functional concerns.

2.2.2 Industrial component models

Today the most common software component models in the industry are Sun's Enterprise Java Beans, .NET and CCM.

2.2.2.1 EJBs

Enterprise Java Beans (EJB) [MIC c] is a specification from Sun Microsystems which addresses the whole lifecycle of component software development, from coding to assembly deployment to the management of running applications. It allows developers to concentrate on the business logic, while non functional services are provided by the container: persistency, transactions, security, load-balancing. EJBs target large, transaction-intensive enterprise scenarios where scalability[4] is a key factor.

[4]*scalability* here means a large number of clients for the application

The specification process follows a concerted path between involved members of the specification committee (Java Community Process [MIC d]) and version 3 has recently been finalized, notably introducing annotation-based configuration.

2.2.2.2 COM and .NET

Microsoft's approach for component-based programming is a pragmatic one and proposes global frameworks[5] rather than strictly defined component models. Common Object Model (COM) [MIC a] was the proposal fostered by Microsoft for software components in the 90's, and is still used in the forthcoming version of Windows, Vista. COM is a binary standard not tied to any particular language. It defines how components interact on a "binary" level by specifying a set of key services, though it does not specify what a component or an object is. COM is underpinned by three fundamental concepts:

- uniquely identified and immutable interface specifications,
- uniquely identified components that can implement multiple interfaces,
- a dynamic interface discovery mechanism.

There is one fundamental entity: the *interface*. A COM entity may propose several interfaces, though one is mandatory: the `IUnknown` interface, which identifies the entire COM entity. DCOM extends COM by adding distributed communication capabilities, and takes charge of the low-level details of network protocols. The COM and DCOM technologies served as a base for many Microsoft technologies, and are now being superseded by the .NET technology [MIC b]. .NET is the new advocated development framework from Microsoft and it provides a broader set of services and interoperability, notably through web services.

2.2.2.3 CCM

The CORBA Component Model (CCM) [GRO] is defined by the Object Management Group, an industrial consortium aiming at defining standards for distributed systems, and the author of the CORBA 1 and CORBA 2 specifications. The CCM is a specification for business components which may be distributed, heterogeneous, and independent from the programming language or operating system. It relies on the previous CORBA specification, hence is based on the notion of distributed objects.

The objectives of the CCM are fist to allow the design of application using a component based approach, with a clear separation between functional and non-functional concerns, and second to specify the lifecycle of component application. Therefore, CCM defines a complete process for the definition, production, deployment and execution of the components. The role of each actor of the process is clearly identified: designer, implementer, packager, deployer, end-user.

A CCM component can be represented as a black box with several access points, as illustrated in figure 2.2. It offers a base reference on itself (the *component reference*). Functional services are defined in the form of *facets* (for receiving invocations) or *sinks* (for receiving events through a message bus). *Receptacle* interfaces are client interfaces requiring which need to be connected with matching facets. *Sources* and *sinks* are interfaces for event-based communications.

[5] a global framework such as .Net provides a large body of pre-coded solutions to common program requirements, and manages the execution of programs written specifically for this framework

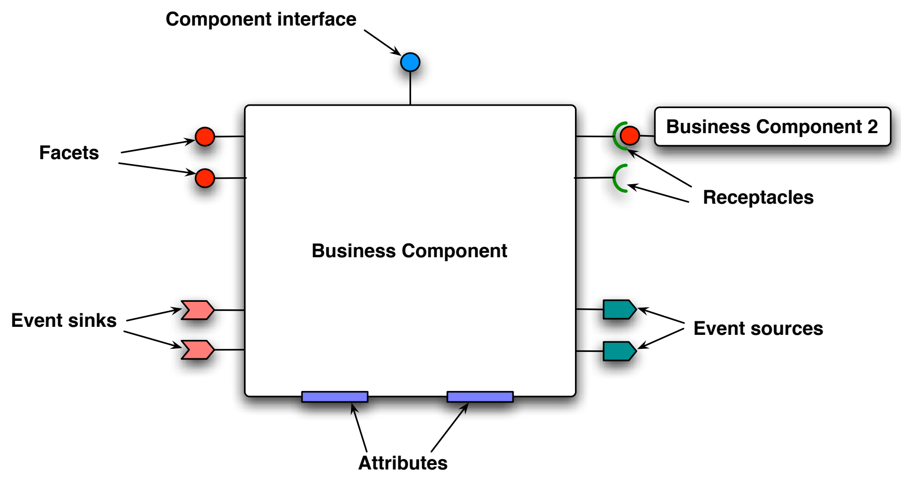


Figure 2.2: *A CCM component*

Component are deployed into component *containers* and created and managed by *homes*. Only the business code needs to be written, all other services are provided by the framework and require configuration.

2.2.3 Academic research in component models

There are many component models proposed within the academic community. Compared to industrial models, they usually do not define a global specification of the lifecycle of the component from development to deployment, as CCM does for instance, and they usually focus on specific points of research interest.

2.2.4 ODP

The Reference Model for Open Distributed Processing (RM-ODP) is a set of standards defined by the International Standards Organization, in an effort to standardize distributed computing [ISO 95]. The ODP standards describe the protocols and interfaces that distributed systems should possess, and they aim at being both abstract enough (for having different implementations) and directive enough (so that the implementations may cooperate).

ODP distinguishes five points of view for specifying distributed systems: the point of view of the *company* (rights and obligations of the software entities); the point of view of the *information* (application semantics); the point of view of the *processes* (a *functional* view with a definition of the interactions between entities); the point of view of *engineering* (which services are required from the infrastructure and how); and the point of view of the *technology* (imposed constraints on the infrastructure).

The point of view of the processes defines the ODP component model: an application consists of components (processing objects) that encapsulate an internal state and data. The internal state may be modified by processing the data, and processing operations are gathered into interfaces. Interactions between components only happen through interfaces, which are typed, and therefore impose typing compatibility for connecting interfaces of components, through a *binding* operation.

Although it defines the foundations of a component model and a high-level view of the lifecycle of the development of applications, ODP does not propose any implementation.

The core concepts of the component model were nevertheless reused for the Fractal and OpenCOM models described next.

2.2.5 OpenCOM

OpenCOM [CLA 01] is a component model proposed by researchers in Lancaster University. OpenCOM aims at reusing the component-based techniques within middleware platforms. It defines a lightweight, efficient and reflective component model that uses the core features of Microsoft COM to underpin its implementation; these include the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the `IUnknown` interface. The higher-level features of COM, including distribution, persistence, transactions and security are not used. Specifically targeting the design of middleware platforms, OpenCOM was concretely used for the implementation of an efficient version of the OpenORB reflective middleware, developed at Lancaster, hence offering support for building high-performance reflection functionality.

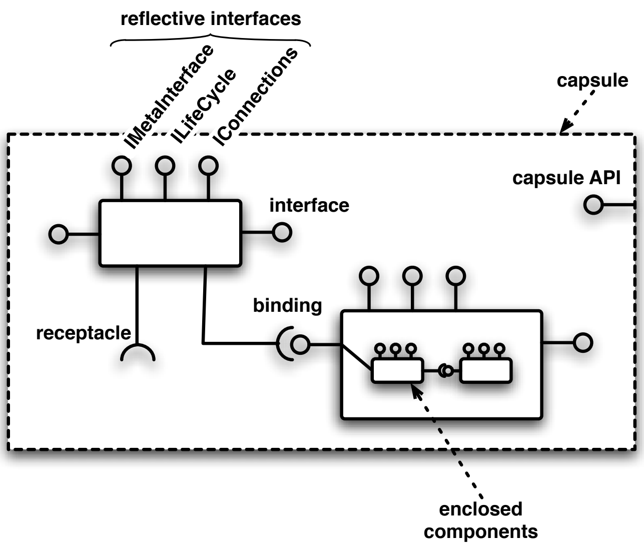


Figure 2.3: *OpenCOM concepts*

The key concepts of OpenCOM are *capsules*, *components*, *interfaces*, *receptacles* and *bindings*, represented in figure 2.3. Capsules are runtime containers and they host components. Each component implements a set of custom interfaces and receptacles. An interface expresses a unit of service provision, a receptacle describes a unit of service requirement and a connection is the binding between an interface and a receptacle of the same type.

OpenCOM is used in various middleware projects, and more recently, in the GridKit middleware [GRA 04], which aims at providing an adaptive middleware for dealing with changing environments, notably for Grid and pervasive computing.

For the moment, OpenCOM remains oriented towards middleware construction, as shown by the recent efforts on building overlays frameworks. It is does not target end-user application designs, however it follows common principles with the Fractal component model that we describe later in this section.

2.2.6 SOFA

SOFA [PLA 98] stands for SOFtware Appliances; it is a component model proposed by Charles University in Prague. It initially targeted the issues of dynamic updates of components, and is now also used for the behavioral verification of component systems, as communications between components can be formally captured.

In the SOFA component model, applications are viewed as a hierarchy of nested components. Components can be either primitive or composed - a composed component is built of other components, while a primitive component contains no subcomponents. As a result, all functionality is located in the primitive components.

A component is described by its *frame* and its *architecture*. The frame is a black-box view of the component. It defines *provides* interfaces and *requires* interfaces of the component. Optionally, it declares properties for parameterizing the component. The frame can be implemented by more than one architecture. The architecture of a composed component describes the structure of the component by instantiating direct subcomponents and specifying the subcomponent's interconnections via interface *ties*. The architecture reflects a particular grey-box view of the component - it defines the first level of nesting in a component hierarchy. A tie which connects two interfaces is realized via a connector. A *connector* is the mediator of an interaction between components, and establishes the rules that govern component interaction [SHA 93]. Connectors have protocol specifications defining their properties. For convenience, the simple connectors, expressing procedure (method) calls are implicit so that they do not have to be specified in an architecture specification.

The whole application's hierarchy of components (all levels of nesting) is described by an application deployment descriptor. Components are bound through connectors, which are first-class entities in SOFA.

A particularity of SOFA is behavior protocols are central to the model, to the point that typing relationships are defined by behavior protocols.

2.2.7 Fractal

Fractal is a modular and extensible component model proposed by INRIA and France Telecom, that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms or graphical user interfaces. Fractal consists of: a general model and a detailed specification with an API [FRAa], a reference implementation called Julia [BRU 04a], and a set of tools and reusable components (an Architecture Description Language, a set of component for remote communications (FractalRMI), a set of tools for management (FractalJMX) and a Graphical User Interface (FractalGUI)).

Since the first version of the specification was proposed in 2002, an active community of academics and industrials has been building around Fractal, as can be observed from the number of available implementation of Fractal and the number of publications experimenting with Fractal, for instance in ECOOP'06 Fractal Workshop [FRAc].

The main features of the model are:

- *composite* components, offering a uniform view of applications at various levels of abstraction,
- *shared* components, to model resources and resource sharing while maintaining encapsulation,
- *introspection* capabilities, in order to monitor a running system,

- *intercession* and reconfiguration capabilities, to *dynamically* configure a system.

Fractal enforces separation of concerns between functional and non-functional features, by distinguishing functional and non-functional (control) access points to components.

The Fractal model is somewhat inspired from biological cells, where exchanges between the content of a cell and the environment are controlled by the membrane. By analogy, a Fractal component, as represented in figure 2.4 is a runtime entity, which offers server and client functional interfaces, as well as control interfaces (for non functional properties) implemented as *controller* objects in the membrane. All interactions with the component are interactions with the membrane of the component, and this allows interception and intercession using interception objects positioned within the membrane.

The Fractal model is an open component model, and in that sense it allows for arbitrary classes of controllers and interceptor objects, including user-defined ones. It is therefore possible to customize the non-functional features of the component. A basic set of controllers is already defined, for setting attributes (`AttributeController`), binding components (`BindingController`), adding components within composite components (`ContentController`) or controlling the lifecycle of the component (`LifeCycleController`).

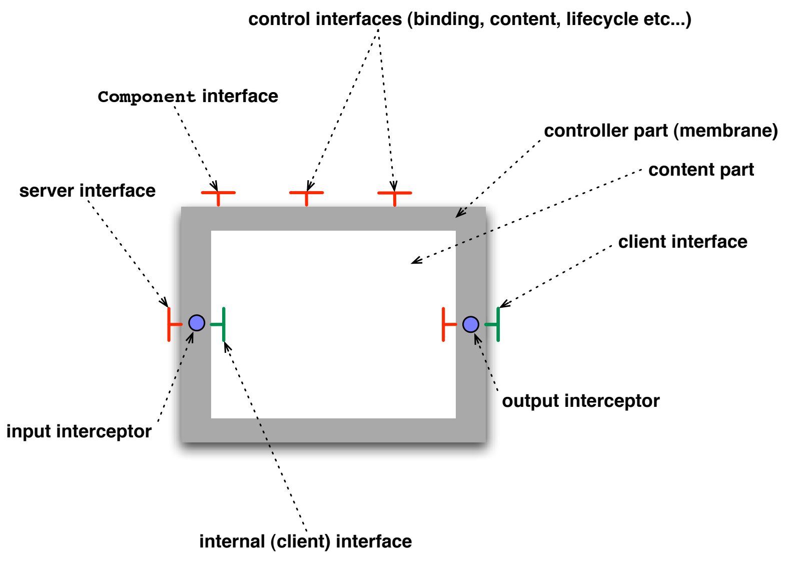

control interfaces (binding, content, lifecycle etc...)
Component interface
controller part (membrane)
content part
server interface
client interface
output interceptor
input interceptor
internal (client) interface

Figure 2.4: *A Fractal component*

Functional interfaces are defined by a *name*, a *role* (client or server), a *cardinality* (singleton or collection), a *contingency* (mandatory or optional) and a *signature* (in Java, the fully qualified name of a Java interface). In the type system proposed by Fractal, the ensemble of functional interfaces defines the type of a component. A Fractal component also provides a `Component` interface, which is similar to the `IUnknown` interface in the COM model, and which allows the discovery of all external client and server interfaces. A Fractal component may be a composite component, in which case it usually contains other components, or it may be a primitive component, which is an atomic component implementing the business logic. The instantiation of components is performed through configurable *factories*.

Components communicate through their interfaces; a client interface is bound to a server interface so that two components can communicate. Fractal supports *primitive* *bindings* and *composite bindings*. A primitive binding is a binding between one client interface and one server interface of compatible types in the same address space. A composite binding is a communications path between an arbitrary number of component interfaces, and it consists of an assembly of primitive bindings and binding components (stubs, skeletons, adapters, etc.). The concept of binding in Fractal is related to the concept of connector in other component models such as SOFA. A typical assembly with bindings and a composite component is presented in figure 2.5

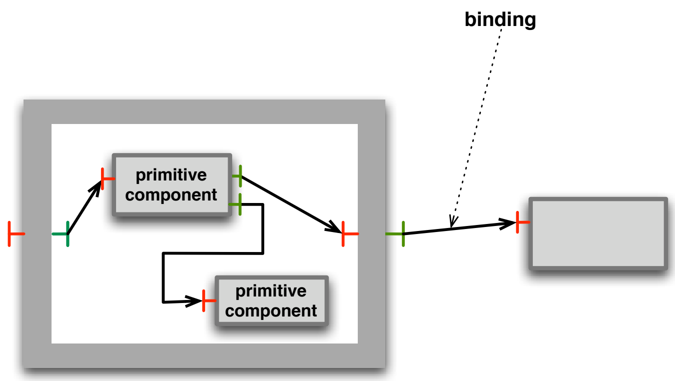


Figure 2.5: *A hierarchical assembly of Fractal components*

Fractal also defines an Architecture Description Language for the description of component systems. FractalADL is an open and extensible language to define component architectures. Associated tools allow the parsing and analysis of ADLs and the deployment of the corresponding component systems. FractalGUI is a graphical tool which allows the creation and graphical visualization of component assemblies.

As Fractal provides a detailed specification and an API, several projects implementation of the Fractal model have been developed as answers to specific needs or for experimenting novel implementation approaches. For instance, FracNet is an implementation of Fractal on the .NET platform, AOKell [SEI 06] is an implementation using Aspect Oriented Programming, Think [FAS 02] is a C implementation targeting the design of operating systems.

The reference implementation is a Java implementation called Julia, and was developed by France Telecom. Julia has four main objectives: first, provide a framework for programming controllers; second, provide a continuum from static configuration to dynamic configuration (a static configuration is more efficient but not reconfigurable); third, provide a highly optimized implementation in order to minimize the time overhead of the framework; fourth, provide an implementation which may be used in constrained environments such as KVM or J2ME virtual machines. Distributed communications are provided through Java RMI binding components. Julia is used in several project and served as a basis for implementing several middleware applications. For instance, Speedo is a JDO (Java Data Object, a specification for the persistency of objects) implementation implemented with Julia. Another interesting example is DREAM (Dynamic REflective Asynchronous Middleware) [LEC 04]. DREAM is a software framework dedicated to the construction of asynchronous middleware. It provides a component library and a set of tools to build, configure and deploy middleware implementing various asynchronous communications paradigms such as message passing, event-reaction, publish-subscribe etc.

2.3 Component models for Grid computing

Common component based programming models and frameworks as listed above provide many facilities for the development of complex and robust applications, however they do not answer the requirements of Grid programming regarding high performance computing, parallelism, dynamicity, interoperability and large-scale deployments.

For these reasons, researchers started investigating component models that would address these Grid computing requirements. In this section we evaluate three of these component models and frameworks: CCA, ICENI and GridCCM.

2.3.1 CCA

CCA stands for Common Component Architecture and is a general component model driven by the CCA Forum. The CCA Forum is an initiative of the Department of Energy in the United States and gathers various DoE national laboratories and collaborating academic institutions. The goals of the CCA Forum are to specify a *minimal component architecture for high-performance computing*, and to *federate research efforts and resources* in this area. The need for performance and for the federation of resources positioned the CCA community as a major actor within the Grid community.

The CCA proposal aims at facilitating inter-software and inter-implementations collaboration by specifying software modules as components, with strictly defined client and server interfaces. This eases the development of multi-physics applications, which require the coupling of independent applications that usually do not exhibit standardized interfaces. Instead of assembling libraries into a monolithic application difficult to maintain and to modify, the CCA model proposes to isolate independent software elements into components with well-defined interfaces. Interconnection of components is standardized and exposed ports are normalized in a Scientific Interface Description Language (SIDL) file.

The CCA proposal consists of:

- A general model, specified in [ARM 99] and [CCA], which defines the core concepts: *component* (the software entity), *framework* (the container) and *ports* (the access points to the component).
- A framework, which is a workbench or container for building, connecting and running components.
- A scientific IDL, which is the actual CCA specification: it describes interactions of components with the framework, and it is a basis for language interoperability.
- A configuration API for interacting with the container.
- A repository API for searching components in a repository.
- A tool for language interoperability and re-use, Babel [KOH 01], which uses IDL techniques to generate glue code between components which normally cannot interoperate.

Components are assembled at runtime by scripts or programs that interact with the CCA frameworks, which act as containers and provide the glue for binding components together, as represented in figure 2.6. A particularity of the CCA proposal is that connections are a dynamic, run-time activity. Ports can be added, removed, and connected at run-time, and this is considered normal behavior.

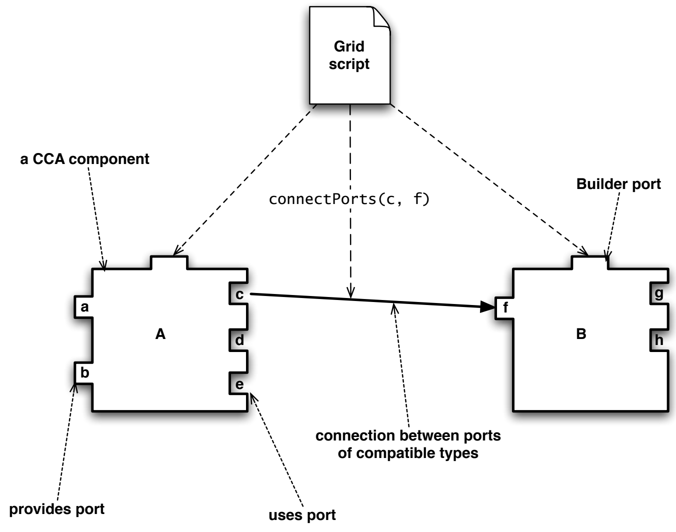


Figure 2.6: *An assembly of CCA components*

A driving idea of the CCA proposal is to address the needs of the High Performance Computing community with the introduction of abstractions for parallelization. Section 2.4.3.4 studies this topic more thoroughly.

There are various implementations of the CCA specification, each tailored for specific needs:

- XCAT [KRI 04] is an implementation where components are deployed as Grid services and interact through Grid services.
- CCAFFEINE [ALL 02] targets Single Program Multiple Data (SPMD) applications and fosters a Single Component Multiple Data (SCMD) model approach, characterized by distributed memory and message passing communications. The implementation relies on a minimal set of functionalities, which may be extended using specialized service components.
- SciRun2 [ZHA 04] is a scientific problem-solving environment (PSE) that allows the interactive construction and steering of large-scale scientific computations. It is based on an existing PSE infrastructure (SciRun [PAR 98]) and on the CCA specification, so that it may interact with other CCA implementations. It supports distributed computing and defines parallel components which use the MxN data distribution mechanism.
- DCA [BER 04] is a CCA framework based on MPI which exhibits MxN data redistribution capabilities.
- Mocca [MAL 05] is a CCA-compliant framework implemented on top of the H2O metacomputing framework [SUN 02], which provides an infrastructure for building Grid environments, in a fashion similar to ProActive.

A strong point of CCA is that it is a model initiated by experimental scientists and targeted for experimental scientists, with a broad community of users. The consequences are that CCA focuses on solving the problems which are relevant to its community, and that many reusable components have been developed for various scientific domains: chemistry, accelerator beam dynamics, fusion, material science etc. [KUM 06]

The flexible nature of CCA ports and connections is supposed to facilitate the integration of CCA components in Problem Solving Environments (PSEs), in which the end user directly manipulates component connections to solve the particular problem at hand. However CCA falls short of providing standardized conception and monitoring tools because these tools usually rely on typed assemblies of components such as those defined in ADLs. The lack of static typing also hinders the verifications which may be performed at design-time. Besides, CCA does not offer a formal specification like the ones of CCM or Fractal. Finally, component assemblies are only flat: composite components containing other components are absent of the programming model.

2.3.2 ICENI

ICENI stands for Imperial College e-Science Networked Infrastructure. It is a Grid programming framework which proposes a component model and a deployment framework, and provides a service-oriented architecture. ICENI explicitly handles both spatial and temporal composition. On one hand, components are assembled in a spatial composition, which is the typical model for component based systems: component client interfaces are linked to other components server interfaces. On the other hand, ICENI also provides temporal composition, which corresponds to workflow description of service-oriented architectures, and can provide useful information for application scheduling. The temporal composition is inferred from the user defined spatial composition by using a graph-based workflow language.

The definition of a component includes metadata about its behavior and implementation requirements.

The lifecycle of an ICENI application is illustrated in figure 2.7: the metadata of the components is used both to create a structural composition and a workflow orchestration (emphasized shapes). Then the deployment is performed thanks to a deployment framework which allows an automatic distribution of components according to *user or quality of service constraints* and infrastructure information, and is interfaced with various middleware for the deployment process: Globus, Condor and Sun Grid Engine among others [YOU 03].

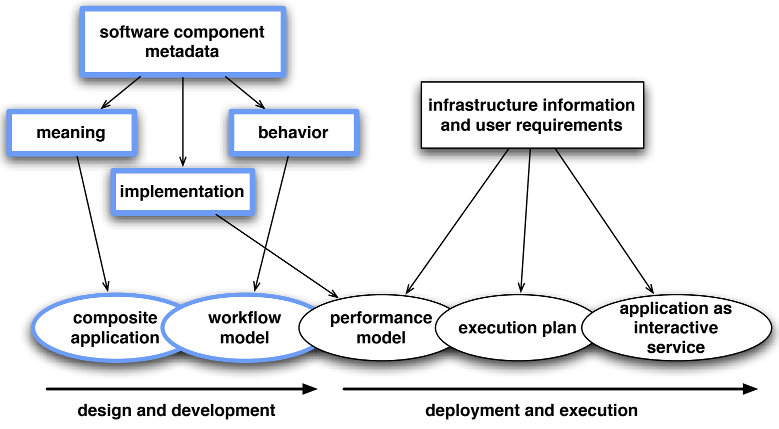


Figure 2.7: *The lifecycle of an ICENI application (adapted from S. Newhouse's presentation at ICENI workshop'04)*

Future versions of ICENI should be developed on top of Web Services, and the architecture decomposed into separated composable toolkits, with third-party component reuse in mind [MCG].

Unfortunately, the development of ICENI is currently stalled due to reorganizations at Imperial College.

2.3.3 HOC-SA

HOC stands for Higher-Order Components. It is a programming model in which first class entities are parameterizable and deployable software units.

HOC-SA (Higher-Order Component Service Architecture) is an execution environment for HOCs. The HOC-SA framework aims at simplifying the development of Grid applications by proposing pre-packaged standard computational patterns and by transparently handling deployment.

HOC applications are created by composing and deploying reusable patterns (skeletons) of parallelism. The patterns are available to grid application programmers as a collection of Web Services. A component in the HOC-SA framework is a software unit that may be deployed on Grid machines using Globus standards, that may be configured by application-specific code or data, and that is accessed through web services. The framework also features code mobility through a code downloading service.

The HOC-SA framework relies on Globus components and is currently being incubated in the Globus consortium.

2.3.4 GridCCM

Research work on providing parallelization for Corba objects (PACO++) [PÉR 04] and Corba components (GridCCM) led to the specification of parallel components used in GridCCM [DEN 04]. The objective of GridCCM is to ground on CCM capabilities (support for heterogeneity, open standards and deployment model) and extend CCM to allow an efficient encapsulation of SPMD codes. Deployment and specification of the components follow the standard CCM specification, notably for the description of components and the deployment process.

GridCCM defines a notion of parallel components, which are CCM components that embed parallel codes. A parallel component can execute in parallel all or some parts of the services it provides. The parallelism of the component is described in a separate configuration file, which contains a description of the parallel methods of the component, the distributed arguments of these methods and the expected distribution of the arguments. Unfortunately, to our knowledge, the implementation of GridCCM is not yet finalized, and the different configurations for parallel invocations are not exhaustively specified.

Collective communications between parallel components are described in more detail in section 2.4.3.5.

2.4 Collective communications

Collective communications designate multiway interactions between software entities.

These interactions are also categorized as *multipoint* communications[6]. Senders

[6]Actually multipoint communications enclose point-to-multipoint, multipoint-to-point and multipoint-to-multipoint interactions

hold references on receivers, enabling *explicit multiway interactions* in the design of applications.

In this section, we evaluate different high-level approaches for achieving collective communications between distributed software entities, as a means to handle the complexity of multiway interactions between numerous *distributed* entities and as a means to improve performance. We are interested in the *parallelism* and *synchronization* features offered by collective communications, therefore we do not expose here communication paradigms such as publish-subscribe or event-reaction. These other features are usually implemented by a tier broker and can be easily integrated in component models by defining send and receive interfaces.

2.4.1 Rationale

We distinguish two ways of achieving collective communications offering parallelism and synchronization features. The first way is to provide *a set of primitives* or connectors for defining 1-to-n and n-to-1 interactions. The second way is to establish an optimized schedule of direct communications between m components and n components, for addressing the so-called *MxN problem*.

In the first way, thanks to a set of primitives for defining collective communications, designers can simply and comprehensively express multiway interactions between components in the system. Data distribution can be customized, and collective communications primitives are a basis for optimizing communications between parallel components. Furthermore, collective operations, as a higher and more sophisticated programming abstraction than simple operations or send-receive operations, allow a better structuration of communications, by notably bringing simplicity, programmability and expressiveness to programs [GOR 04].

The second way is related to the MxN problem, illustrated in figure 2.8, and defined as follows:

Definition 2.4.1 *The MxN problem refers to the problem of communicating and exchanging data between parallel programs, from a parallel program that contains M processes, to another parallel program that contains N processes.*

The parallel programs in the MxN problem are usually SPMD programs (Single Program Multiple Data). In order to reach maximum efficiency, frameworks that specifically address the MxN problem perform *direct connections* between parallel entities of the parallel programs, so that data is directly exchanged and does not suffer superfluous copies. The connections between distributed entities are determined by *communication schedules*, which define the sequences of messages required to correctly move data among cooperating processes. Computing communication schedules is usually a complex operation, which may be facilitated by the use of external dedicated libraries.

One of the objectives of this thesis is to define a consistent and well defined programming model for collective interactions between components, which can be further extended and implemented to tackle the MxN problem. In the remaining paragraphs of this section, we present the different approaches for collective communications, starting with message passing and invocations. We then focus on collective communications in component models.

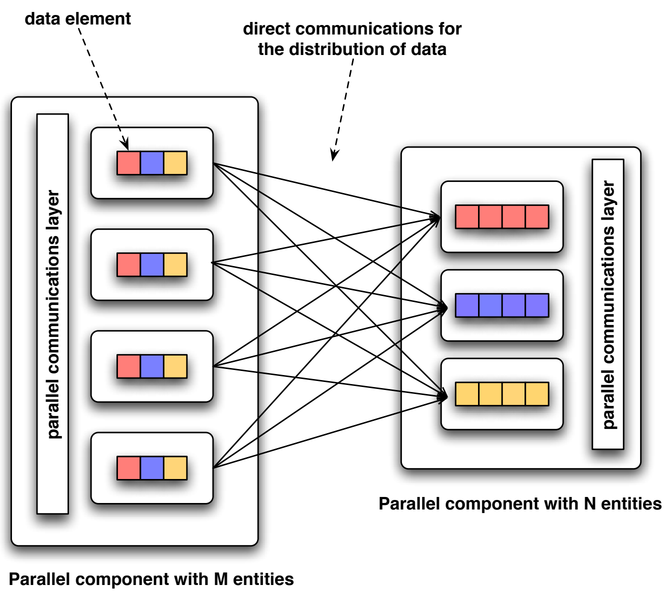

data element
direct communications for the distribution of data
parallel communications layer
parallel communications layer
Parallel component with N entities
Parallel component with M entities

Figure 2.8: *The MxN data redistribution problem*

2.4.2 Common models and frameworks for collective communications

We consider two main approaches for multipoint communications: message-based, and invocation-based.

2.4.2.1 Message-based collective communications

PVM and MPI are the two main standards for programming parallel and distributed applications communicating by exchange of messages.

PVM provides a virtual abstraction of a parallel machine, although processes are distributed. Multipoint communications are achieved with direct addressing of with process groups, but they are not very efficient. MPI (Message Passing Interface) is a standard for the SPMD programming model, using message passing as a communication paradigm [MPI94]. It is commonly used for scientific applications, and it defines some collective communication operations. The processes participating to a collective communication are identified in a *group communicator*. Group communicators allow a dynamic selection of processes participating to a collective communication. Some collective operations in MPI address message distribution: a *broadcast* operation sends the same message to a group of processes, and a *scatter* operation sends a distinct part of a given message to each member of a group of processes. Some collective operations address message gathering: a *gather* operation gathers the messages sent from a group of processes, and a *reduce* operation combines messages sent from a group into a structure, according to a predefined reduction operation. Another collective operation addresses synchronization: a *barrier* operation blocks the processes of a given group until they all have reached the barrier.

On top of these basic operations for collective communications in MPI, many algorithms have been developed for optimizing performance in complex systems, such as the recursive-doubling or dissemination algorithms.

MPI proved a good approach for a large number of users, and one may highlight several features applicable to other forms of collective communications: the various communication modes available, the basic nature of available reduction operations, and the dynamicity provided through group communicators. Messaged-based communications with MPI however remains a low-level paradigm, complex to code and hard to debug.

2.4.2.2 Invocation-based collective communications

Collective communications can be performed using *group method invocations*: this allows a high-level design of group communications, and parallelism can be provided by the implementations. Group invocations correspond to the invocation of a given method on a group of objects. This requires the *identification of the group* and the *specification of the distribution of parameters*, if parameters are to be decomposed between invocation receivers.

We now discuss some of the frameworks that provide group method invocations.

- Various frameworks were built on the CORBA Object Request Broker, using either a transparent integration approach requiring modification of the ORB, like Orbix+Isis and Electra [LAN], or a service approach, requiring explicit manipulation of groups (examples include a Group Communication Service [FEL 96]). Other investigations intended to efficiently encapsulating Single Program Multiple Data codes in CORBA objects, in order to facilitate the coupling of parallel applications (hence addressing the aforementioned MxN problem). such as Cobra [PRI 98], PACO [REN 99] or PARDIS [KEA 97]. These approaches define *parallel objects* as sets of identical sequential objects, and the broker is responsible for transferring *distributed arguments* between inner sequential objects of remote parallel objects (as in figure 2.8). The parallelism is defined at the level of the IDL, and an official OMG specification also formalizes this notion [GRO 03]. PACO++ [PÉR 04] leverages the PACO experiment by aiming at a portable specification of parallel objects (i.e. without modification to the IDL or to the ORB). It considers parallelism as an implementation issue, and parallelism is therefore specified outside of the IDL. Proxies are responsible for the management of parallelism, and this approach does not require modifications to the ORB.
- The Group Method Invocation framework (GMI) [MAA 02] allows a high flexibility in the management of invocations, but delegates the management of the groups to the programmer. Indeed, the programmer has to implement specific interfaces: a member of a group must implement the `GroupInterface` interface, and the customization of results handling and method invocations also require the implementation of specific interfaces. The GMI framework is flexible but forces the programmer to add non-functional code (managing group communications) to the objects members of the group. Besides, objects that do not implement group interfaces cannot be members of a group.
- ProActive Typed Groups [BAD 05a] provide asynchronous group invocations in a transparent manner. A typed group is a reference on a collection of objects of the same type, and when methods defined by the group type are invoked on the group, invocations are performed in parallel using multithreading, and FIFO ordering is guaranteed. Parameters are distributed among group members in a *broadcast* or *scatter* manner. Unfortunately a scatter distribution only occurs for parameters that are groups, in which case the elements of these groups are distributed according to one-to-one mapping between the i^{th} element of a group parameter and the i^{th}. Moreover, scatter distribution mode is restricted to a one-to-one mapping, and

if the sizes of the parameter group and of the target group are different, the distribution proceeds with the following policy: if the parameter group is bigger than the target group, the excess members are *ignored* (therefore all parameters are not distributed); if the parameter group is smaller that the target group, then the members of the parameter group are *reused* in a round-robin fashion (the number of parameters that are distributed varies). Extensions to ProActive Typed Groups were proposed in order to address some of these issues [BAD 05c].

Unfortunately, to our knowledge, frameworks for multipoint collective communications based on invocations only provide one-to-n communications, and *do not provide n-to-one or m-to-n communications*. To our understanding, MxN communications in CORBA parallel objects actually correspond to 1-to-n communications initiated from individual objects contained in parallel objects: there is no gathering of invocations.

2.4.3 Collective communications in component models

Few component models explicitly define semantics of collective communications. They rely on messages, invocations, and intermediate components.

2.4.3.1 Connection-oriented programming

In [SZY 02], Szyperski describes *connection-oriented programming* as a programming paradigm for linking software entities but avoiding statically chained call dependencies, using indirections that can be configured at runtime. The connection oriented programming paradigm is used in the Fractal model, through an inversion of control pattern used by the binding controllers. The author also considers one-to-many connections and outlines that intermediate abstractions - channels or groups - can be introduced to facilitate connections to sets of components. These groups or channels correspond to connectors, or binding components in the Fractal terminology, that decouple the caller-callee connection, and they can provide useful services such as filtering, count, delay etc.

On one hand, Szyperski, mentions connections from several callers to one callee as possible and even common. But on the other hand, he does not explicitly consider many-to-one connections with their capabilities: synchronizations of callers invocations, and reduction or gathering of invocation parameters.

2.4.3.2 ODP

The binding model of ODP defines two kinds of bindings: *primitive bindings* and *compound bindings*. Primitive bindings correspond to direct bindings beetwen interfaces of compatible types, while compound bindings correspond to indirect bindings through binding objects (like Fractal's composite bindings), as represented in figure 2.9. In ODP, binding objects are fully configurable and the semantics associated with the bindings are customizable: different communication models (event based, synchronous etc ...) can be used, and one can specify multiway bindings.

Some of these concepts are implemented in the Dream project [LEC 04], which provides a set of binding components for Fractal.

We believe that instead of relying on explicit intermediate binding components, which complexifies the design, it is possible to define some of the semantics associated with multiway bindings at the level of the interfaces themselves, through the definition of collective interfaces.

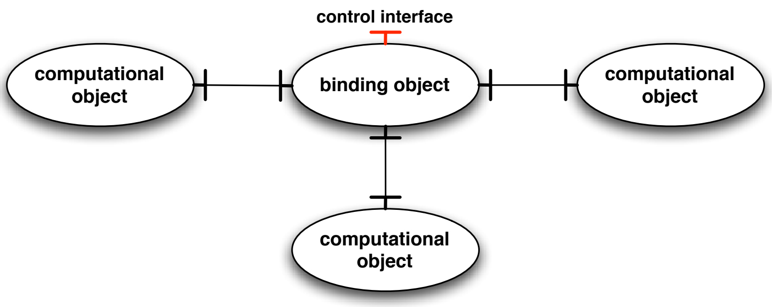


Figure 2.9: *ODP compound binding*

2.4.3.3 ICENI

ICENI defines collective communications between multiple interfaces. They are achieved through the use of *tees*: binding objects that are generated automatically from configuration information. Collective communications in ICENI were actually designed for the distribution of XML data. Similarly to the binding components of ODP, the tees allow the specification of different communication policies between multiple interfaces. The different kinds of tees are represented in figure 2.10. For instance, *switch* tees connect

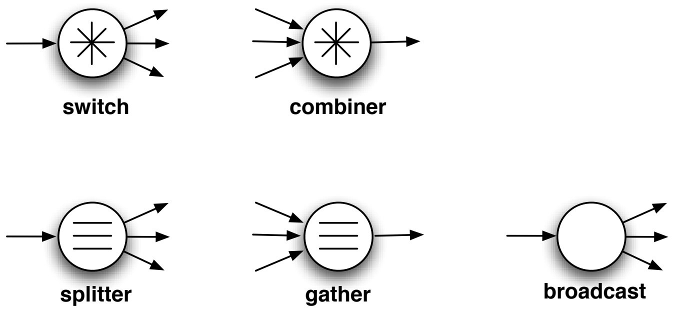


Figure 2.10: *ICENI tees*

one client interface to several server interfaces, and can dispatch messages from the client to selected server interfaces. Inversely, *combiner* tees can redispatch messages from several selected client interfaces to one server interface, and combiner tees include a buffering mechanism to store incoming data. *Splitter* tees scatter data from one client interface to several server interfaces, while inversely, *gather* tees combine data from several clients, using a buffering mechanism, and transfer the resulting data to a single server interface. *Broadcast* tees forward a given invocation from a single client interface to several server interfaces, with the same parameters. Tees have already proven useful in the Grid Enabled Integrated Earth system model of the UK e-Science project [MAY 03].

Unfortunately, tees only perform data redistribution for arrays, and only work on XML data. The ICENI tees are therefore similar in functionality to ODP binding components.

2.4.3.4 CCA

Collective method execution and data redistribution between parallel components are addressed by several groups within the CCA community, notably through the Parallel Application WorkSpace (PAWS) [KEA 01], the CUMULVS project [GEI 97], the PRMI specification [DAM 03]; a sum-up of these works is available in [BER 05]. Two approaches are undertaken in CCA for parallel data redistribution. The first one is to encapsulate parallel data redistribution support into a standard component, which can then be composed with other components to realize different coupling scenarios. The second one is to use Parallel Remote Method Invocation for embedding the parallel data redistribution.

The first approach is the one adopted in PAWS. PAWS introduces the notion of *collective port* for CCA as an intent to let the component framework in charge of the collective interactions and let the developer focus on higher-level program design. A collective port designates one logical connection in the design, but is implemented as multiple network connections. The behavior of the collective ports is customizable in terms of remote invocations and of data redistribution between interacting components. Collective interactions between parallel components are handled through intermediate *translation components* (also called MxN components), in charge of the redistribution of invocations and data from M components to N components. Translation components are composable into higher level translation components, facilitating the management of complex interactions. The Seine framework [ZHA 06] also defines MxN components with a virtual-geometry shared space approach and efficient local computation of communication schedules.

The second approach relies on collective remote invocations defined in the Parallel Remote Method Invocation (PRMI) specification as proposed in [DAM 03]. PRMI is used in DCA [BER 04], which is a framework for distributed collective communications with CCA, tied to MPI. This framework chose the second approach described later for parallel data redistribution. For dynamically selecting participating components of a collective invocation, DCA uses MPI communicator groups passed as the last argument of collective method invocations.
Although there has been some intents to precisely define collective interfaces in the CCA specification, to our knowledge this specification is not exhaustive, only simple cases are mentioned, and there are no details on the redistribution features; this raises many questions on the use and capabilities of such interfaces.

2.4.3.5 GridCCM

GridCCM aims at coupling parallel codes by providing direct communications between parallel components and optimally distributing data. An example of coupled GridCCM components is given in figure 2.11 and we can observe it is very similar to the illustration of the MxN problem in figure 2.8: optimized internal communications with MPI or PVM, and external communications through CORBA.

The parallelism of applications is described in an auxiliary XML file, which contains a description of the parallel methods of the component, the distributed arguments of these methods and the expected distribution of the arguments. Coupling is realized by non-functional entities of the parallel component; they perform the connection bindings and compute the communication schedules for the redistribution of data, possibly using external data redistribution libraries.

An abstract parallel programming model has been defined for GridCCM[RIB 04]. It is articulated around four aspects. First, it defines what are parallel and distributed entities: a collection of distributed entities, some of them managing the business aspects, usually SPMD codes, and the others managing the non functional aspects. Second, it defines how parallel entities communicate, which includes the connection phase, the communication phase itself and the deconnection phase. Third, it addresses the distribution of data between parallel entities. Fourth, it considers the management of exceptions between distributed parallel entities.

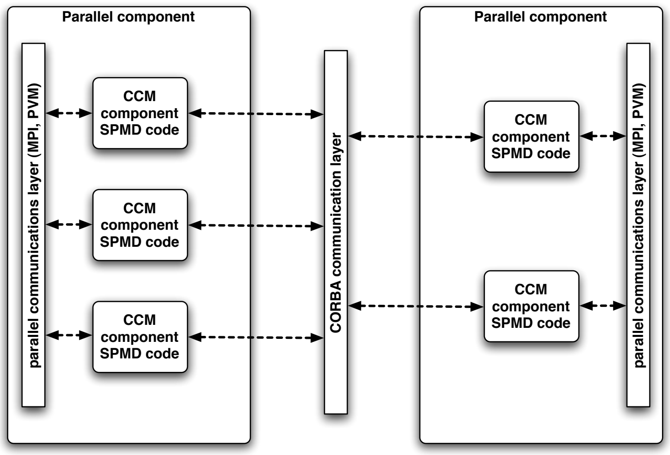


Figure 2.11: *Coupling of parallel components with GridCCM*

GridCCM's approach for the MxN problem results in efficient couplings, but seems tedious to design and requires a deep knowledge of each coupled application.

2.5 Discussion

In this state of the art, we justified our choice of a component-based programming approach with respect to other programming approaches, and we described the general expected benefits. We presented existing standard component models, pointed out that they do not address the requirements of Grid computing, and finished by evaluating component models specifically addressing Grid computing. We also presented different approaches to collective communications.

A component model suitable for Grid computing should present good properties as a general component model, so that it can be extended and adapted to correspond to the many diverse use cases of Grids. We propose an overview of these properties in table 2.1 for the component models we presented.

On a general scope, industrial models tend to have complex specifications (marked as not lightweight in the table) because they address in detail the whole development and deployment process, including packaging issues. On the other hand, academic models tend to be more specialized. For example OpenCOM is oriented towards reflectivity and adaptation, while SOFA is oriented towards behavioral specifications. Academic models are usually more easily extensible, because they are not proprietary nor require a complex standardization process. Table 2.1 also highlights the general properties of the Fractal model, which is not surprising as Fractal was indeed designed to be a general component model. On the contrary, CCA cannot be characterized as a general purpose component model, because it lacks core features such as standardized description or reflection.

Properties of component models	.Net	EJB	CCM	OpenCOM	SOFA	Fractal	CCA	Iceni	GridCCM	ODP
Lightweight	-	-	-	-	✔	✔	✔	-	-	✔
Extensible	-	-	✔	✔	✔	✔	✔	✔	✔	✔
Hierarchical	-	-	-	✔	✔	✔	-	-	-	✔
Standard composition description	?	✔	✔	✔	✔	✔	-	✔	✔	-
Non-functional adaptivity	-	-	-	✔	-	✔	-	-	-	-
Reflective	?	-	-	✔	✔	✔	-	-	-	✔
Explicit API	✔	✔	✔	✔	?	✔	✔	✔	✔	-
Dynamic reconfiguration	-	-	✔	✔	✔	✔	✔	✔	✔	✔
Sharing	-	-	-	-	-	✔	-	-	-	-
Packaging / repository	✔	✔	✔	?	✔	?	?	✔	✔	?

- no ✔ yes ? unspecified or unknown

Table 2.1: *Evaluation of the general properties of evaluated component models*

A component model suitable for Grid computing should also answer the requirements of Grid computing: this is studied in table 2.2, where we consider the issues of deployment, heterogeneity, interoperability, high-performance, parallelism (and collective communications) and scalability[7], presented in the introduction of the document.

Industrial component frameworks are not oriented towards Grid computing because they lack parallel features, high performance codes encapsulation, and they do not manage entities on numerous remote locations. Furthermore, some of these frameworks, such as EJBs or .Net, are usually designed to run on a single administrative domain.

Considering how the component models we presented fulfill Grid requirements, the most adequate candidates for a component model for Grid computing are GridCCM, CCA and ICENI, however, table 2.1 shows that they are not satisfactory as general purpose component models. In particular, CCA lacks static typing, which prevents the use of ADLs; GridCCM does not offer hierarchical structuration, sharing, or reflective capabilities; ICENI components do not offer reflective features nor hierarchical designs, and they depend on a complex framework. For these reasons, we propose to use Fractal, the most general model according to table 2.1, as a basis for this work, and to define extensions to fulfill the missing requirements of Grid computing.

2.6 Context: a model based on active objects

ProActive is an open source Java library for Grid computing. It allows concurrent and parallel program and offers distributed and asynchronous communications, mobility, and a deployment framework. With a reduced set of primitives, ProActive provides

[7]Here, *scalability* must be understood as the ability to handle a large number of distributed entities.

Requirements of Grid computing	.Net	EJB	CCM	OpenCOM	SOFA	Fractal[5]	CCA	Iceni	GridCCM	ODP
Deployment framework	✔	✔	✔	–	✔	?	✔	✔	✔	–
Multiple deployment protocols	N/A	–	✔	–	–	?	?	✔	✔	–
Heterogeneity[1]	–	✔	✔	✔	✔	✔	✔	✔	✔	–
Interoperability	✔ WS	✔ WS	✔ CORBA	?	?	?	✔ CCA – CCA	✔	✔ CORBA	–
Legacy software	✔	✔	?	?	?	?	✔	✔	✔	–
HPC[2]	–	–	–	–	–	–	✔	✔	✔	–
Collective communications[3]	–	–	–	–	–	–	✔ PRMI	✔ XML data	✔	–
MxN facilities	–	–	–	–	–	–	✔	–	✔	–
Scalability[4]	–	–	✔	?	–	–	✔	✔	✔	–

– no ✔ yes ? unspecified or unknown

[1] hardware and operating system
[2] targets high performance computing applications
[3] excluding publish-subscribe or queuing mechanisms, where sender does not know the receivers
[4] capability to handle a large number of distributed entities
[5] Fractal implementations, except ProActive/Fractal

Table 2.2: *Fulfillment of Grid computing requirements in studied component models*

an API allowing the development of parallel applications which may be deployed on distributed systems and on Grids.

The active object model and the ProActive library are used as a basis in this thesis for developing a component framework and performing large scale experiments.

2.6.1 Active objects model

ProActive is based on the concept of *active object*, which can be seen as a fine to medium-grained entity with its own configurable activity.

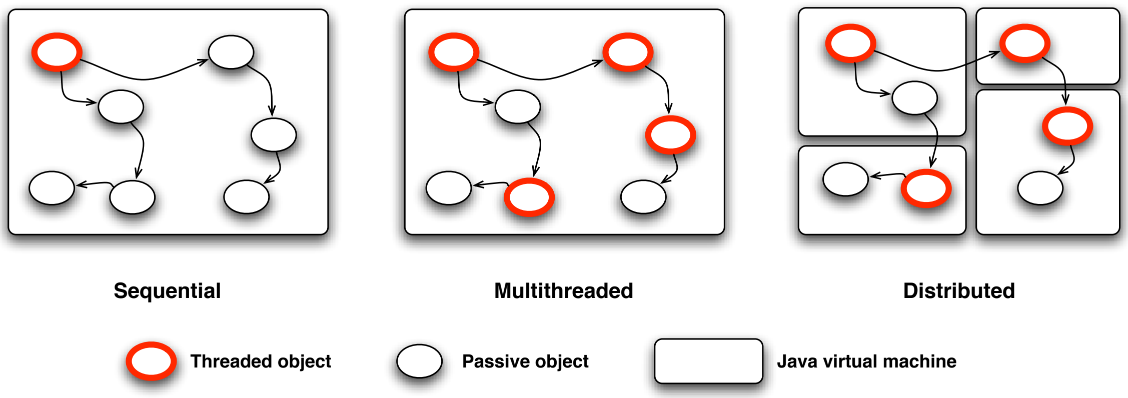


Figure 2.12: *Seamless sequential to multithreaded to distributed objects*

A distributed or concurrent application built using ProActive is composed of a number of active objects (figure 2.12). Each active object has one distinguished element, the root, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous[8] with transparent future objects and synchronization is handled by a mechanism known as wait-by-necessity [CAR 93]. A future is a placeholder for the result of an invocation, which is given as a result to the caller, and which is transparently updated when the result of the invocation is actually computed. This whole mechanism results in a data-based synchronization. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee, in order to ensure causal dependency.

Explicit message-passing based programming approaches were deliberately avoided: one aim of the library is to enforce code reuse by applying the remote method invocation pattern, instead of explicit message-passing.

The active object model of ProActive guaranties determinism properties and was formalized with the ASP (Asynchronous Sequential Processes) calculus [CAR 04].

2.6.2 The ProActive library: principles, architecture and usages

The ProActive library implements the concept of active objects and provides a deployment framework in order to use the resources of a Grid.

2.6.2.1 Implementation language

Grids are inherently heterogeneous environments, which usually gather several flavors of processors and operating systems. In this context, using a language which relies on a virtual machine allows maximum portability. ProActive is developed in Java, a cross-platform language and the compiled application may run on any operating system proposing a compatible virtual machine. Moreover, ProActive only relies on standard APIs and does not use any operating-system specific routine, other than to run daemons or to interact with legacy applications. There are no modifications to the JVM nor to the semantics of the Java language, and the bytecode of the application classes is never modified.

2.6.2.2 Implementation techniques

ProActive relies on an extensible meta-object protocol architecture (MOP), which uses reflective techniques in order to abstract the distribution layer, and to offer features such as asynchronism or group communications.

The architecture of the MOP is presented in figure 2.13. An active object is concretely built out of a root object (here of type B), with its graph of passive objects. A body object is attached to the root object, and this body references various features meta-objects, with different roles. An active object is always indirectly referenced through a proxy and a stub which is a sub-type of the root object. An invocation to the active object is actually an invocation on the stub object, which creates a reified representation of the invocation, with the method called and the parameters, and this reified object is given to the proxy object. The proxy transfers the reified invocation to the body, possibly through

[8]provided they fulfill minimum conditions, such as reifiable return type and no declared exceptions in the method

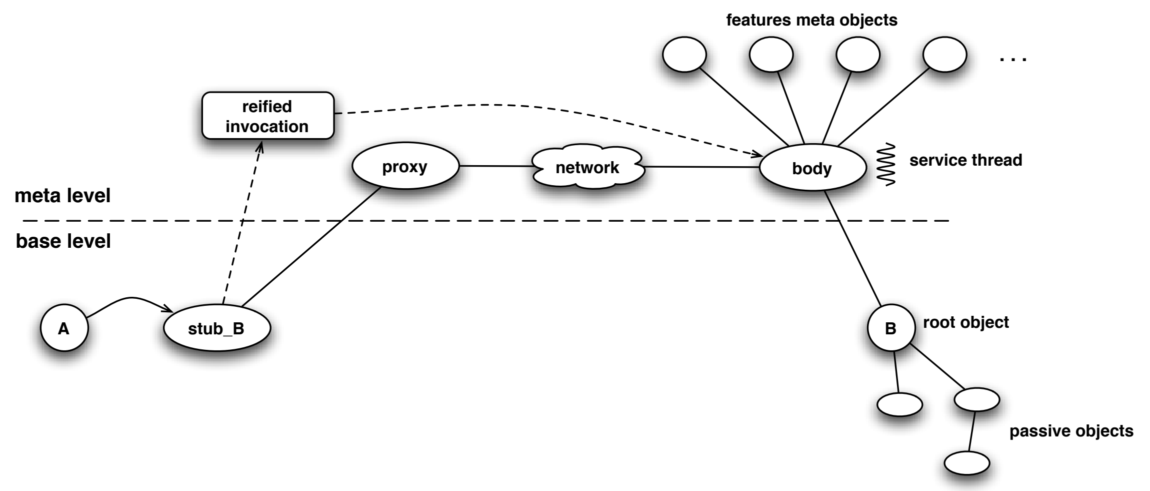


Figure 2.13: *Meta-object architecture*

the network, and places the reified invocation in the request queue of the active object. The request queue is one of the meta-objects referenced by the body. If the method returns a result, a future object is created and returned to the proxy, to the stub, then to the caller object.

The active object has it own activity thread, which is usually used to pick-up reified invocations from the request queue and serve them, i.e. execute them by reflection on the root object. Reification and interception of invocations, along with ProActive's customizable MOP architecture, provide both transparency and the ground for adaptation of non-functional features of active objects to fit various needs. It is possible to add custom meta-objects which may act upon the reified invocation, for instance for providing mobility features, or as we will see in chapter 5, to provide an implementation of a component model.

Active objects are instantiated using the ProActive API, by specifying the class of the root object, the instantiation parameters, and a possible location information:

```
// instantiate active object of class B on node1 (a possibly remote location)
B b = (B) ProActive.newActive(''B'', new Object[] {aConstructorParameter}, node1);

// use active object as any object of type B
Result r = b.foo();

// possible wait-by-necessity
System.out.println(r.printResult());
```

2.6.2.3 Semantics of communications

In ProActive, the distribution is transparent: invoking methods on remote objects does not require the developer to design remote objects with explicit remoting mechanism (like `Remote` interfaces in Java RMI). Therefore, the developer can concentrate on the business logic as the distribution is automatically handled and transparent. Moreover, the ProActive library preserves polymorphism on remote objects (through the reference stub, which is a subclass of the remote root object).

Communications between active objects are realized through method invocations, which are reified and passed as messages. These messages are serializable Java objects which may be compared to TCP packets. Indeed, one part of the message contains routing information towards the different elements of the library, and the other part contains the data to be communicated to the called object.

Although all communications proceed through method invocations, the communication semantics depends upon the signature of the method, and the resulting communication may not always be asynchronous.

Three cases are possible: synchronous invocation, one-way asynchronous invocation, and asynchronous invocation with future result.

- *Synchronous invocation*:
  - the method return a non reifiable object: primitive type or final class:
    `public boolean foo()`
  - the method declares throwing an exception:
    `public void bar() throws AnException`

  In this case, the caller thread is blocked until the reified invocation is effectively processed and the eventual result (or Exception) is returned. It is fundamental to keep this case in mind, because some APIs define methods which throw exceptions or return non-reifiable results. This is the case of the Fractal API, which was implemented during this thesis using the ProActive library.

- *One-way asynchronous invocation*: the method does not throw any exception and does not return any result:
  `public void gee()`
  The invocation is asynchronous and the process flow of the caller continues once the reified invocation has been received by the active object (in other words, once the rendez-vous is finished).

- *Asynchronous invocation with future result*: the return type is a reifiable type, and the method does not throw any exception:
  `public MyReifiableType baz()`
  In this case, a future object is returned and the caller continues its execution flow. The active object will process the reified invocation according to its serving policy, and the future object will then be updated with the value of the result of the method execution.

If an invocation from an object A on an active object B triggers another invocation on another active object C, the future result received by A may be updated with another future object. In that case, when the result is available from C, the future of B is automatically updated, and the future object in A is also update with this result value, through a mechanism called *automatic continuation* [CAR].

2.6.2.4 Features of the library

As stated above, the MOP architecture of the ProActive library is flexible and configurable; it allows the addition of meta-objects for managing new required features. Moreover, the library also proposes a deployment framework, which allows the deployment of active objects on various infrastructures.

The features of the library are represented in figure 2.14.

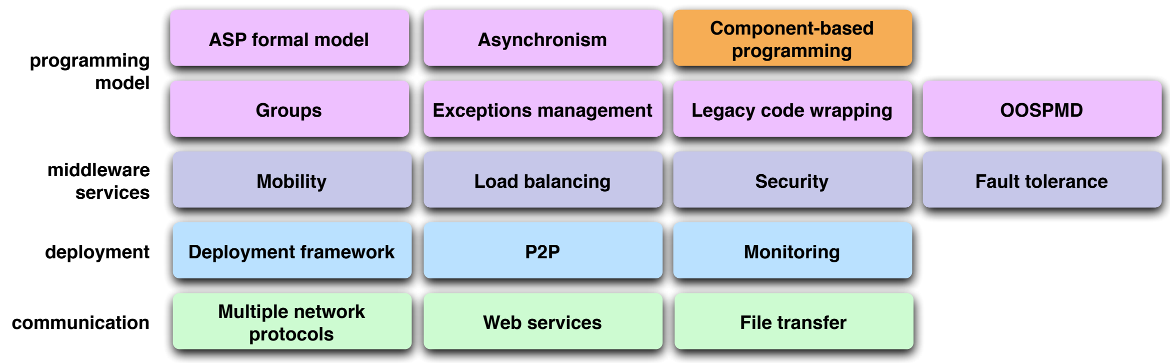


Figure 2.14: *Layered features of the ProActive library*

The active object model is formalized through the ASP calculus [CAR 05], and ProActive may be seen as an implementation of ASP. The library may be represented in three layers: programming model, non-functional features and deployment facilities.

The programming model consists of the active objects model which offer asynchronous communications, typed group communications [BAD 02] and the object-oriented SPMD programming paradigm [BAD 05b]. In the context of this thesis, we propose a new programming model for ProActive: component-based programming.

Non-functional features include a transparent fault-tolerance mechanism based on a communication-induced checkpointing protocol [BAU 05], a security framework for communications between remote active objects [ATT 05], migration capabilities for the mobility of active objects [BAU 00], a mechanism for the management of exceptions, and a mechanism for wrapping legacy code, notably as a way to control and interact with MPI applications.

The deployment layer includes a deployment framework [BAU 02]; it is detailed in the next section, and it allows the creation of remote active objects on various infrastructures. A peer-to-peer infrastructure is also available; it allows the acquisition of objects distributed within the infrastructure, in order to use them for peer-to-peer computations. The load balancing framework uses the migration capabilities to optimize the placement of the distributed active objects.

In the communication layer several protocols are provided for the communication between active objects: Java RMI as the default protocol, HTTP, tunneled RMI. It is also possible to export active objects as web services, which can then be accessed using the standard SOAP protocol. A file transfer mechanism is also implemented; it allows the transfer of files between active objects, for instance to send large data input files or to retrieve results files [BAU 06].

2.6.2.5 Deployment framework

The deployment of Grid applications is too often done manually, using remote shells for launching the various virtual machines or daemons on remote computers and clusters. The commoditization of resources through Grids and the increasing complexity of applications are making the task of deploying central and harder to perform. ProActive succeeds in completely avoiding scripts for configuration, getting computing resources, etc. It provides, as a key approach to the deployment problem, an abstraction from the source code so as to gain in flexibility. We hereby describe the fundamental principles of the deployment framework, and more information and examples are available from the official ProActive documentation [PRO].

Principles A first key principle is to fully eliminate from the source code the following elements:

- machine names,
- creation protocols,
- registry and lookup protocols.

The objective is to deploy any application anywhere without changing the source code. Deployment sites are called nodes, and correspond for ProActive to JVMs which contain active objects. A second key principle is the capability to abstractly describe an application, or part of it, in terms of its conceptual activities. In other words, to abstract away the underlying execution platform, and to allow a source-independent deployment the framework must provide the following elements:

- an abstract description of the distributed entities of a parallel program or component,
- an external mapping of those entities to real machines, using actual creation, registry, and lookup protocols.

XML deployment descriptors To answer these requirements, the deployment framework in ProActive relies on XML descriptors. These descriptors introduce the notion of *virtual node* (VN):

- a VN is identified as a name (a simple string),
- a VN is used in a program source,
- a VN, after activation, is mapped to one or to a set of actual ProActive Nodes, following the mapping defined in an XML descriptor file.

A virtual node is a concept of a distributed program or component, while a node is actually a deployment concept: it is an object that lives in a JVM, hosting active objects. There is of course a correspondence between virtual nodes and nodes: the function created by the deployment, the mapping. This mapping is specified in the deployment descriptor. There is no automatic mapping between virtual nodes and active objects: the active objects are deployed by the application on the infrastructure nodes mapped by a virtual node. By definition, the following operations can be configured in the deployment descriptor:

- the mapping of VNs to nodes and to JVMs,
- the way to create or to acquire JVMs,
- the way to register or to lookup JVMs.

Figure 2.15 summarizes the deployment framework provided by the ProActive middleware. Deployment descriptors can be separated in two parts: mapping and infrastructure. The VN, which is the deployment abstraction for applications, is mapped to nodes in the deployment descriptors, and nodes are mapped to physical resources, i.e. to the infrastructure.

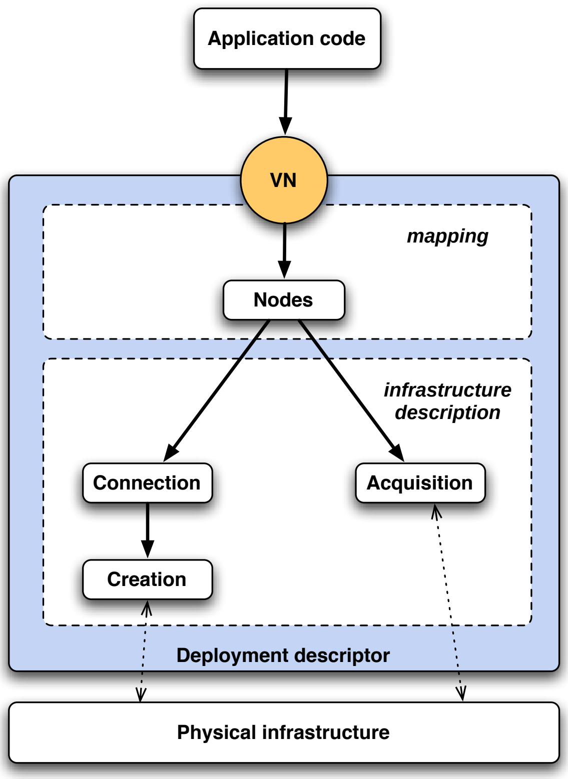


Figure 2.15: *Descriptor-based deployment*

Retrieval of resources In the context of the ProActive middleware, nodes designate physical resources from a physical infrastructure. They can be created or acquired. The deployment framework is responsible for providing the nodes mapped to the virtual nodes used by the application. Nodes may be created using remote connection and creation protocols. Nodes may also be acquired through lookup protocols, which notably enable access to the ProActive peer-to-peer infrastructure.

Creation-based deployment : Machine names, connection and creation protocols are strictly separated from the application code, and ProActive deployment descriptors provide the ability to create remote nodes (remote JVMs). For instance, deployment descriptors are able to use various protocols:

- local,
- ssh, gsissh, rsh, rlogin,
- lsf, pbs, sun grid engine, oar, prun,
- globus (GT2, GT3 and GT4), unicore, glite, arc (nordugrid).

Deployment descriptors allow to combine these protocols in order to create remote JVMs, e.g. log on a remote cluster frontend with ssh, and then use pbs to book cluster nodes to create JVMs on each. In addition, the JVM creation is handled by a special process, localJVM, which starts a JVM. It is possible to specify the classpath, the Java install path, and all JVM arguments. It is in this process that the deployer specifies which transport layer the ProActive node uses. For the moment, ProActive supports as transport layer: RMI, HTTP, RMIssh, Ibis, and SOAP.

Acquisition-based deployment The main goal of the peer-to-peer (P2P) infrastructure is to provide a new way to build and use Grids. The infrastructure allows applications to transparently and easily obtain computational resources from Grids composed of both clusters and desktop machines. The application deployment burden is eased by a seamless link between applications and the infrastructure. This link allows applications to be communicant, and to manage the resources volatility.

2.6.2.6 Large scale experiments and usages

It would be difficult to qualify a middleware as a Grid middleware without demonstrating its Grid capabilities: deployment on a large number of hosts from various organizations, on heterogeneous environments and using different communication and connection protocols.

We outline two series of events which illustrate the capabilities of ProActive for large scale deployments.

First, the n-queens computational problem was solved for n=25 using a distributed peer-to-peer infrastructure on about 260 desktop machines in INRIA Sophia Antipolis, with an idle cycle stealing approach. The n-queens problem is a classical computational problem which consists of finding the placements of n non-attacking queens on a n*n chessboard. It is a NP-hard problem, which means that high values of n require years of computation time on a single machine, making this problem ideal in the context of challenge for Grid computing.

The peer-to-peer infrastructure was highly heterogeneous: Linux, Windows, various JVMs, Pentium II to Xeon bi-pro from 450 Mhz to 3.2 GHz, etc, and the total duration time was slightly over 6 months (4444h 54m 52s 854), starting October 8th until June 11th, using the spare CPU cycles of about 260 machines. The cumulative computing time was over 50 years.

Second, the Grid PlugTests events held at ETSI in 2004 and 2005 demonstrated the capacity of the ProActive library to create virtual organizations and to deploy applications on various clusters from various locations. The first Grid PlugTests event [GPT05] gathered competing teams which had to solve the n-queens problems by using the Grid built by coordinating universities and laboratories of 20 different sites in 12 different countries, resulting in 900 processors and a computing power of 100 Gigaflops (SciMark 2.0 benchmark for Java). In the second Grid PlugTests [GPT] event, a second computational problem was added: the permutation flow-shop, in which the deployment of an optimal schedule for N jobs on M machines must be computed. Heterogeneous resources from 40 sites in 13 countries were federated, resulting in a 450 Gigaflops grid, with a total of 2700 processors.

2.7 Conclusion

In this chapter, we positioned the work of this thesis in the context of Grid computing: we focus on programming models and tools. We demonstrated the validity and adequacy of a component-based approach. We explained why the Fractal component model is the most appropriate candidate for a Grid component model, provided we address the missing requirements for Grid computing. Then we introduced the ProActive Grid middleware.

A programming framework for Grid computing comprises a programming model and design and assembly tools, and a middleware layer to access the Grid infrastructure.

Some of the requirements for Grid computing exposed in chapter 1 should be fulfilled by the middleware layer: distribution, heterogeneity, interoperability and multiple administrative domains. Another part of the requirements should be fulfilled by the programming model and programming tools: complexity. The remaining requirements should be addressed by both the programming layer and the middleware layer: the deployment process relies on a deployment framework provided by the Grid middleware, and on a deployment model provided by the programming model. Dynamicity is addressed by the programming layer, through applicative adaptability, and by the middleware layer, through load balancing and code mobility. Scalability is addressed by the programming layer by providing design facilities for collective interaction and parallelism, and by latency management. High performance depends on both the programming model and the middleware efficiency.

We argue that the Fractal component model and the ProActive middleware can be extended and combined in order to provide a comprehensive framework for Grid computing. In this thesis, we focus on the definition of a programming model based on component-based programming and we describe the general principles of this model in the following chapter.

Chapter 3

A Component Model for Programming Grid Applications

In this chapter, we propose a component model for programming Grid applications, named *ProActive/Fractal*. The proposed model extends the Fractal component model and combines it with an active object model.

In chapter 1, we identified the specificities of Grid computing. A programming model and framework for Grid computing must be able to address these specificities: distribution, deployment (across multiple administrative domains, with security constraints), scalability, heterogeneity, interoperability and legacy software, high performance, complexity and dynamicity.

A programming model based on the component paradigm must also fulfill a number of requirements specific to component models, so that it may suit the various kinds of applications which may be used on Grids.

3.1 Requirements for the component model

In this section, we evaluate the choice of Fractal as a general component model.

We list these requirements as follows:

- It must be *extensible*, in order to add non-functional features or communication paradigms that may be added for specific needs.
- It must be *lightweight*, so that several implementations may be developed following different approaches and objectives.
- It must have *well-defined semantics*, through a clear specification and an API.
- It must provide *extensible system description configuration*, preferably through an ADL, so that graphical tools may be built to help the design of the systems.
- It must provide room for *packaging* mechanisms, so that component systems may be encapsulated and retrieved from repositories.
- It must provide a clear *separation* between functional and non functional concerns.
- It must be *reflective*, so that introspection and intercession techniques may be used for dynamic adaptation and reconfiguration.

In chapter 2, we showed that the Fractal component model fulfills all of these requirements. As a consequence it appears well suited as a general purpose component model which may be specialized to fulfill custom requirements, such as those of a Grid computing framework. We also showed that the Fractal model is more general than CCA or GridCCM. This is why we decided to further investigate the opportunity of using Fractal in the context of Grid computing.

3.2 Adequacy of the Fractal model for the Grid

In this section, we evaluate whether the Fractal component model is a suitable programming model in the context of Grid computing. For each of the requirements we listed for Grid computing in chapter 1, we verify there is no incompatibility with the Fractal model. Table 2.2 showed that Fractal does not currently fulfills or does not address Grid requirements, and we investigate whether Fractal can be extended to answer these issues. In addition, we identify which Grid requirements should be handled by the underlying Grid middleware, and which ones should be handled by an extension of the Fractal model.

3.2.1 Distribution

The Fractal component model does not impose any locality constraint, and rather states that:

> *coupled with the use of meta-programming techniques, component-based programming can hide to application programmers some of the complexities inherent in the handling of non-functional aspects in a software system, such as distribution [FRAa].*

Therefore Fractal components may be distributed entities. These managed entities can be deployed and communicate through an underlying Grid middleware.

Besides, we argue that latencies inherent to large scale distribution can be handled thanks to asynchronous communications and a model which offers a control over the localization of the distributed entities, so that highly communicating components can be closely located, in the same subnetwork for instance. This is related to the deployment capabilities, evaluated in the next paragraph.

3.2.2 Deployment

In the definition of components we adopted in 2.1.2.6, a component is a unit of deployment. We consider deployment as a two-phase process: first, the acquisition of Grid resources, and second, the instantiation and assembly of component systems on these resources.

Although the deployment process is a fundamental part of the lifecycle of a software product, it is not part of the Fractal specification. In a Grid environment, it must be considered thoroughly and addressed in a proposal of a component model for Grid computing.

Fortunately, Fractal proposes a standardized description of component systems in the form of an Architecture Description Language (the Fractal ADL), and this ADL may serve as a basis for the specification of a deployment process.

The Fractal Architecture Description Language allows the specification of component assemblies in a flexible and extensible manner. Therefore it should fit most of the specific needs of Grid applications.

ADL may also be used for ensuring the correctness of a composed system. The type system of Fractal allows syntactic verifications, but semantic verifications are only possible through the specification of behaviours. Fortunately, the extensiveness of the ADL should allow the addition of such behavioral specifications. In [BAR 05] for instance, the idea is to extend the ADL to specify behavioural properties of components, then to generate automatas from both the behavioural specification and the architecture of the system, and finally to use model checking tools to verify the automatas, hence the design.

3.2.3 Heterogeneity

The Fractal model has been implemented in different programming languages (Java [BRU 02, SEI 06], Smalltalk [BLO , BLO , fractalk] C [FAS 02], C++ [LAY 04]), and notably defines an API in Java. The Java language runs on most environments, as demonstrated during the Grid PlugTests events (section 2.6.2.6).

Heterogeneity is therefore not an issue for the Fractal model, it is allowed by the model and handled by the underlying middleware.

3.2.4 Interoperability and legacy software

Some pieces of a Grid application, such as numerical solvers or graphical control interfaces, are highly optimized pieces of software. They are programmed with the language and framework which are the most appropriate for the intended work (for example Fortran/MPI for numerical analysis), and compiled for specific architectures. There are three approaches to integrate such software into a componentized Grid environment.

The first approach is to rewrite the application into the language of the component framework, so that the framework can directly interact with this piece of software. However, this task is: expensive, may not match the performance of the initial software, and can be avoided.

Indeed a second approach is to wrap the legacy application into components which are able to translate invocations on component interfaces to invocations on the legacy software, and vice-versa, for example as proposed in [HUA 03], or with the Babel framework of CCA [KOH 01]. This approach is a first step towards code coupling (for example, an acoustics code with an aerodynamics code, both developped for MPI and wrapped as Fractal or CCA components).

A third approach is to access legacy software through web services: this way interoperability is guaranteed no matter which technology the legacy software is using. In this approach, providers of components expose their components in a standardized way through web services, whereas in the second approach we presented, components are encapsulated in an ad-hoc manner. The Grid community, which is very concerned with interoperability, is promoting the use of web services as a communication protocol between software components in a Grid. This corresponds to a general trend towards service oriented architectures, that are very much advocated these days and well suited for linking coarse-grained components.

These three approaches are valid in the Fractal model, because Fractal allows any programming language, architecture or communication standard.

3.2.5 High performance

In a Grid application, particularly in scientific computing, some parts of the system need high performance in terms of computation and of communications, while other parts, which rely on user input for instance, do not have high-performance requirements.

Computation efficiency usually depends on both the programming language and the hardware resources. Although Java has been proven an efficient programming language for scientific applications [BUL 03] - with performance still improving -, many applications still rely on highly optimized libraries in FORTRAN and use the SPMD programming paradigm. Wrapping legacy code as exposed above is a way to reuse these optimized codes. However, considering that high-performance computing is generally associated with SPMD programming and parallel communications, offering SPMD approaches with parallel communications could be addressed by the component model. Components provide a higher level of abstraction and a more simple way to design SPMD applications, as we will show in section 4.3.

Multi-physics applications usually involve code coupling between parts of the application which perform different specific tasks, but need to communicate efficiently during execution. The MxN problem is an example of such coupling problematics. The Fractal component model proposes the encapsulation of components inside composite components, and communications are not supposed to cross directly the membranes of composites (for direct communications between encapsulated parallel component for instance).

3.2.6 Complexity

We consider three ways of handling the complexity at the level of the component model: hierarchical composition, sharing, and collective communications.

3.2.6.1 Hierarchical composition

Most component models, including those used in the context of Grid programming (e.g. CCA, GridCCM or ICENI), only allow flat compositions. We advocate that a hierarchical structuration is a valuable means to tackle the complexity of Grid systems, by allowing the definition and organization of sub-systems, and we outline two clear advantages.

First, hierarchical structuration facilitates the *identification of architectural units* of various granularities. Grid systems indeed are usually composed of many different and heterogeneous software units. Each software unit may itself integrate other software units.

An illustration is given by the NAS Grid benchmarks (NGBs) [FRU 01b], a set of benchmarks specified by the NASA, which are designed to be representative of some typical usages of Grids (regarding the needs of the NASA). They use some applications specified by the NAS parallel benchmarks (NPBs). NPBs define a set of kernels, which are typical computational problems, and a set of applications that use some of these kernels. NGBs are designed as data flow graphs whose nodes are very coarse-grained computation units. These nodes encapsulate applications specified in NPBs, and that may individually run on a whole dedicated cluster. As represented on figure 3.1, several levels of hierarchies are clearly identifiable: this kind of design is very well suited for a hierarchical component composition.

Second, this hierarchical structuration facilitates the *identification of the roles of human actors* upon the system. For example, the end user of an experiment is interested

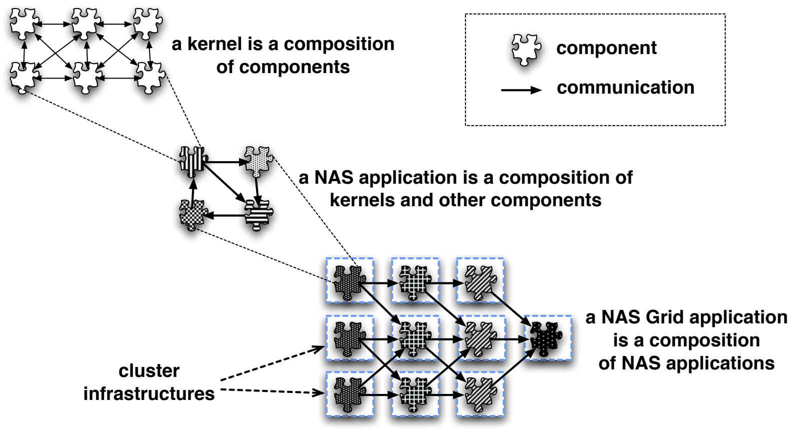

a kernel is a composition of components
component
communication
a NAS application is a composition of kernels and other components
a NAS Grid application is a composition of NAS applications
cluster infrastructures

Figure 3.1: *A conceptual view of the hierarchical structuration of a NAS Grid benchmark*

in the top level functionnalities, and see the application as one big piece of software. The user of a Grid portal wants to define or customize her application by composing high-level subsystems in a workflow. A developer trying to optimize the access to databases within a computational unit focuses on a specialized subsystem.

Hierarchical composition presents some drawbacks: hierarchies can be extremely costly in the context of highly distributed applications. Geographically speaking, a composite component may have its membrane hosted on a given host, but the components it contains can be located on different sites. If those contained components are themselves hierarchical components, they may also present the same kind of geographical distribution. This means that for a given message to eventually reach its target, it may have to cross several membranes, and therefore pay for the cumulative cost of each remote communication between membranes. Because this is not acceptable, shortcut facilities should be provided in a distributed context. Another solution could be to use sharing, presented in the next paragraph. For instance, sharing could help address the MxN problem presented in 2.4.1 by providing direct communications between encapsulated or wrapped components.

3.2.6.2 Sharing

A Fractal component can be shared among several components. The Fractal specification suggests using shared components as "undo" actions of graphical user interfaces, for common loggers, or for the representation of resources. Grid applications can benefit from sharing some of their components, particularly when these are exposed as Grid services, that can be accessed from virtually anywhere. Grid services are defined in the Open Grid Services Architecture (OGSA) [FOS 02]. In this architecture, each service is offered as a web service, and Grid services include data services (management of replicated copies, query execution and updates, transformation of data into new formats), security services (enforcement of security policies within a virtual organization, authentication and authorization of users), computational services (scheduling of units of work on the hardware infrastructure) or messaging services (subscription to event notifications from other applications or from the infrastructure). While the OGSA defines a programming model based on services, it was originally based on the OGSI (Open Grid Services Infrastructure) as the underlying infrastructure. OGSI has now evolved into WSRF (Web Services Resource Framework) [WSR04], a lighter and clearer specification which is closer to web services and defines stateful services. In the Fractal model one could conveniently represent such stateful services as shared components.

In the model we propose based on active objects, sharing is possible however the ProActive/Fractal framework *does not yet implement sharing*.

3.2.6.3 Parallelism

Parallelism is a key feature in Grid computing. In order to use most of the available computing power, (long) computations are usually parallelized: a master component sends simultaneously a set of tasks to a set of remote components, that will process these tasks in parallel. Parallelism is usually used in specific parts of a Grid application, which require high performance and perform parallel computation on dedicated machines (or peer-to-peer infrastructures), but it may also be generalized within the application, for building parallel workflows for instance, and for managing a large number of distributed entities.

Collective communications - one-to-many, many-to-one or many-to-many communications - are a way to express parallelism, but they are not considered in the Fractal specification, and rather delegated to binding components such as in the DREAM framework. We consider collective communications as a fundamental paradigm to handle complexity and provide parallelism programming facilities, and therefore we propose in chapter 4 an extension of the Fractal specification that defines collective communications *at the level of interfaces*.

3.2.7 Dynamicity

When a system is constituted of a large number of machines (Grid systems range from a few to several thousand hosts), and when computations can last for days, failures must be considered. These failures are not bugs of the application: they are bugs of the underlying operating system, or hardware failures. In some cases, parts of the system are not available anymore, because for example the connection to some clusters is lost, or because all the machines were busy, the jobs cannot be run in the place and time they were scheduled, and a timeout is reached. The system needs to be able to reorganize to cope with these kinds of failures. This implies detecting the problem (this needs to be addressed by the application functional or non-functional code), and then replacing or redirecting the bindings to the components that failed. The dynamic capabilities of the Fractal model can help tackle these kinds of issues. Besides, reconfigurations can also help the system to adapt to the environment, which may be evolving over time: a cluster may be overloaded, in which case the work would better be done somewhere else, or an end user may want to change the configuration during runtime (to change the dataset of a given computation for example).

More generally, in Fractal foundation paper [BRU 02], the authors identify mobility (in the sense of dynamicity) as a requirement for component models:

> *Containment and resource dependencies among components should be allowed to change and evolve over time, whether spontaneously or in reaction to interactions amongst components or between components and their external environment. Such a facility is crucial to avoid architectural erosion, whereby run-time system structures diverge from architecture descriptions, and to model complex reconfiguration processes.*

Dynamicity is therefore a strong requirement, and we want to combine the benefits of hierarchical structuration with the benefits of dynamicity.

3.2.8 Conclusion and role of the underlying middleware

As we have shown in this section, the Fractal model is a programming model that can be adapted for Grid computing. It can directly or indirectly (through extensions), address the issues of: system description, distribution, scalability, interoperability, complexity and dynamicity.

We identified two main areas where the Fractal component model must be enhanced or specialized for Grid computing: *deployment* and *collective communications*.

The role of the underlying middleware on which the Fractal model relies is fundamental in the context of Grid computing, as this middleware provides the distribution (across administrative domains) , deployment and non-functional services which are specific to Grid computing.

In the next section, we define a specialization of the Fractal model based on the active object model, which fulfills the identified shortcomings of the current Fractal component model.

3.3 A component model based on the active object model

We propose a specialization of the Fractal component model in which a component is an active object and as such inherits from the properties of the active object model. Our model is articulated around four core concepts: components are independent activities, communications are asynchronous whenever possible, collective communications are defined at the core of the programming model, and a deployment model is provided. Interoperability is not addressed in this proposal, however it may be easily achieved by exporting component interfaces as web services.

3.3.1 A component is an active object

In our model, a component is constituted of at least one active object, and may contain other active objects, either as internal references in the case of a primitive component, or as enclosed components in the case of a composite component.

A composite component does not define any functional activity, and a primitive component may redefine its functional activity, as what can be done with standard active objects. The functional activity of a component is encapsulated: it starts when the lifecycle of the component is started and stops when the lifecycle is stopped.

Because Fractal components may be compositions of components and active objects may have variable levels of granularity, components in ProActive/Fractal may be considered as *distributed entities of variable granularity*. In that sense, they fit in the definition proposed by Laforenza in [LAF 02]:

> *a Grid programming model can be seen as the cooperation of a set of coarse-grained communicating processes.*

Furthermore, these coarse-grained entities may themselves be constituted of distributed entities, as in the NAS Grid applications, and the ProActive/Fractal model also addresses the design of these coarse-grained entities.

ProActive/Fractal components do not share memory and they interact through typed method invocations. Components receive incoming invocations as reified invocation objects which are queued and processed in a *sequential and non-preemptive* manner, according to a configurable policy (FIFO is the default and recommended one for components).

A component in our model may be seen as an independent configurable activity, which may be moved around for balancing or colocation purposes, and which is remotely accessible in a transparent way.

3.3.2 Communication paradigm

Key challenges for Grid computing applications are to master the network latency and the scalability. The Fractal model opens different strategies to overcome this challenge. One of them is to use binding components to implement sophisticated message-passing mechanism, such as publish/subscribe or pulling. Such mechanisms are useful when defining compositions in time (workflows), where the logic is driven by a sequence of events. Another strategy is to use asynchronous invocations, which also making possible an overlapping between computation and communications, and which can be realized without binder components.

We provide the second strategy in our model: inter-component communications are typed method invocations through component references and may be asynchronous and offer data-based synchronization capabilities (wait-by-necessity, first class futures).

3.3.2.1 Asynchronous and synchronous communications

As specified in 2.6.2.3, communications are asynchronous if the invoked method does not declare throwing any exception *and* does not return a primitive type or a final class.

In the Fractal API, most control methods declare throwing exceptions, which means that *invocations on standard controllers are synchronous*, and one must be careful in a context of hierarchical components with independent activities, in order to avoid deadlocks. We later study the implications on dynamic reconfiguration in this context.

3.3.2.2 Optimizations

We propose two optimizations for the communications between components.

The first one tackles the problem mentioned earlier of invocations crossing several membranes before reaching a given component, and therefore paying the cost of possible distributed communications. In the case that communications are not intercepted, then it is possible to provide short cuts within a context of asynchronous communications and distributed components. This feature is provided in our implementation and described in chapter 5. One implementation principle that we followed is to use a *tensioning* mechanism by extending the rendez-vous of the invocation so that causally ordered communications are still guaranteed. This optimization however comes at the expense of dynamicity of the component assembly, similarly to the compromises due to short cuts made in the Julia [BRU 04a] synchronous and local implementation.

The second optimization is actually an extension of the programming model: collective communications, and more specifically in the context of optimizations, *multicast* communications with parallel invocations. The next chapter provides detailed information on the topic.

3.3.3 Deployment model: a virtualization of the infrastructure

The deployment phase is fundamental in the component-based programming paradigm. It consists of several activities, and usually involves first a packaging activity, where all artifacts and configuration descriptions are integrated (release step). Related activities involve the configuration of the system and components so that the components can be instantiated on selected resources, and the selection and allocation of suitable resources.

3.3.3.1 General model

The deployment model is based on:

- the description of the application using the Fractal ADL,
- the abstraction of the physical infrastructure through the concept of virtual nodes,
- the mapping of the virtual infrastructure to the physical infrastructure.

A simplified representation of the general deployment process is given in figure 3.2 and features three main roles: the *designer*, the *integrator* and the *deployer*. The designer/developer creates the code and assembles components with an ADL. The integrator gathers the code and description of components into packages. The components exhibit virtual nodes names, and the deployer is responsible of mapping these virtual nodes on the available physical infrastructure.

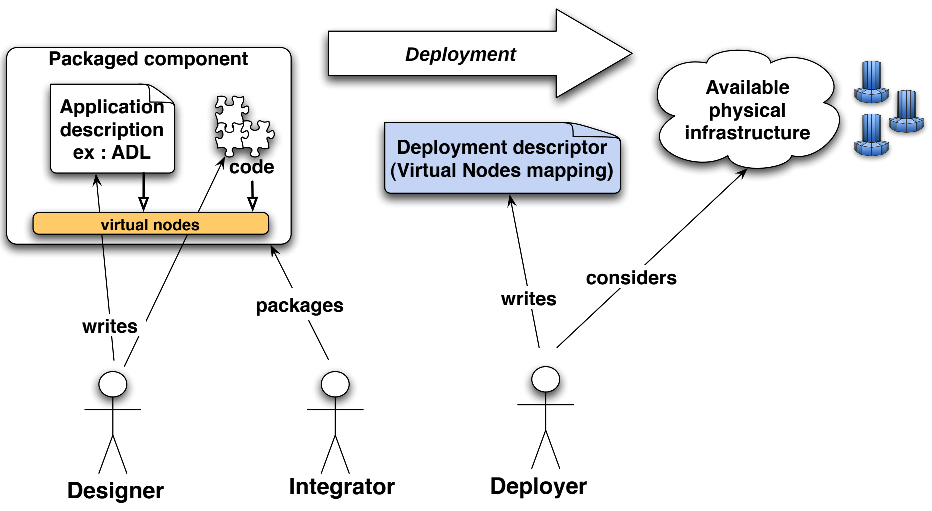

Packaged component
Application description ex : ADL
code
virtual nodes
Deployment
Deployment descriptor (Virtual Nodes mapping)
Available physical infrastructure
writes
packages
writes
considers
Designer
Integrator
Deployer

Figure 3.2: *Overview of the component deployment process and roles*

The deployer may be assisted by deployment tools for the discovery and selection of the most suitable resources, according to user constraints (that may be specified in the description of virtual nodes, as we proposed in [CAR 06]). Therefore this deployment model is compatible with other proposals that, based on the application design, the user constrains, the resource static information, and the resource runtime information, establish a deployment plan for an optimal allocation of the resources; examples include ICENI (figure 2.7), GrADS [BER 01], and in [COP] the authors describe a common approach between existing deployment planners for ASSIST and GridCCM, respectively named GEA [DAN 05] and Adage [LAC 04b].

3.3.3.2 Cardinality of virtual nodes

Virtual nodes also have a *cardinality* property which may be either *single* or *multiple*. If it is multiple, then the virtual node must be mapped to several nodes on the deployment infrastructure. A primitive component located on a multiple virtual node is - unless explicitly specified otherwise - instantiated on each of the underlying nodes. This is a convenient way to deal with multiple instances, and it can be combined with a more explicit design using parameterizable patterns in the ADL (see section 7.2).

Composed virtual nodes inherit the cardinality property: a virtual node composed of at least one virtual node of cardinality multiple is itself multiple. The multiplicity property is verified at deployment time, when performing the mapping of the components.

As we demonstrated, the composition of virtual nodes allows a control over the distribution at design-time. We envisage in a future version to constrain the deployment by specifying constraints on the virtual nodes themselves [CAR 06]. These deployment constraints may be used by resource traders, matchmaking of resources such as in Condor [RAM 98] and automatic deployment frameworks[1] such as GADe [LAC 04a], or ICENI's deployment framework. As a result, combining both design-time and deployment-time control over distribution will ensure an optimal distribution of the components within the Grid.

Further investigations on deployment in the context of Grid computing include pattern-based deployment and are discussed in chapter 7.

3.3.3.3 Controlled deployment through the composition of virtual nodes

The mapping of the virtual nodes to the physical infrastructure can be done with colocation or on the contrary disjoint location purposes. Unfortunately the deployer who is in charge at this step may not be aware of the architectural specificities of the component system, necessary for correctly applying such kind of optimizations. This is why we propose the *composition of virtual nodes* as a means to control the distribution of the components at the design level. This mechanism may also benefit from cost-based scheduling strategies based on static or runtime analysis of the Grid resources and particularly of the latency such as in [DUM 06].

Deployment abstractions can be included within the ADL, such as virtual nodes. Virtual nodes are virtual locations of the components, that are later mapped to an available physical infrastructure (see 2.6.2.5). This allows a given component configuration to be deployed on different infrastructures, without having to modify the design of the components. Moreover, these deployment abstractions can help tackle performance issues through coallocation or disjoint allocation of nodes to components. Coallocation usually makes components closer: this is beneficial if some components communicate a lot together and if these communications are a performance bottleneck because of the latency of the network. As a disclaimer to the difficulties of distributed computing [WAL 94], researchers at Sun Microsystems noted that:

> *Ignoring the difference between the performance of local and remote invocations can lead to designs whose implementations are virtually assured of having performance problems [...] A properly designed application will require determining, by understanding the application being designed, what objects can be made remote and what objects must be clustered together.*

[1]Automatic deployment frameworks try to optimize deployments based on application requirements and infrastructure information.

In Grid computing these considerations are valid and applicable at different scales: communicating entities may be located on the same machine, on the same dedicated cluster, on the same subnetwork, or widely distributed.

In order to provide control *at design time* over the deployment of components that are themselves made of other components, we extended the ADL so that virtual nodes are also composable. This allows co- or disjoint- location strategies when assembling existing components.

We assume the assembly of components is described in an ADL[2] (although the same concepts may apply for a programmatic description): each component is associated to a named virtual node, for instance here the primitive component `A` is associated the virtual node `VN1`.

```
<definition name="A">
   <content class="AImpl"/>
   <controller desc="primitive"/>
   <virtual-node name="VN1"/>
</definition>
```

Components must *export* their virtual nodes so that virtual nodes can be composed when the component is integrated within a broader system. As a result, the description of a component system exposes all the virtual nodes used in the system, and they have been composed in the most adequate manner by designers of enclosed components.

Exportation of virtual nodes is defined in the ADL as in the following example, which describes the exportation of the virtual nodes VN(1) and VN(2) in the description of the composite component F:

```
 <exportedVirtualNodes>
      <exportedVirtualNode name="VN(1,2)">
         <composedFrom>
            <composingVirtualNode component="B" name="VN(1)"/>
            <composingVirtualNode component="C" name="VN(2)"/>
         </composedFrom>
      </exportedVirtualNode>
</exportedVirtualNodes>
```

The composition is actually a renaming which is performed by the designer of the enclosing system definition. The designer may optimize the organization of component within a virtual architecture, so that highly communicating component may be colocated for instance (colocated either on the same host, or on the same cluster for instance).

An example of virtual node composition is given in figure 3.3: we can see that virtual nodes are renamed, and that VN(1), VN(2), VN(3) and VN(4), independently defined in the ADLs of composing components, are eventually mapped on the same infrastructure, for instance here a cluster of machines.

Figure 3.4 shows a corresponding component system which exports virtual nodes. Thanks to the mapping, all the components B, C, D, E, F, G and H, which are part of the same subsystem, are mapped on the same virtual node. Therefore, they can be

[2]In order to keep ADL files coherent with the code of primitive components, it is possible, during the development phase, to automatically generate ADL files from annotations in the code, thanks to standardized annotations and an annotation processing framework called *fraclet-annotation* [ROU 06], to which we contributed by beta-testing and providing feedback. Fraclet-annotation also automatically creates the code required by the Fractal framework for applying dependency injection; this helps separating functional from non-functional code.

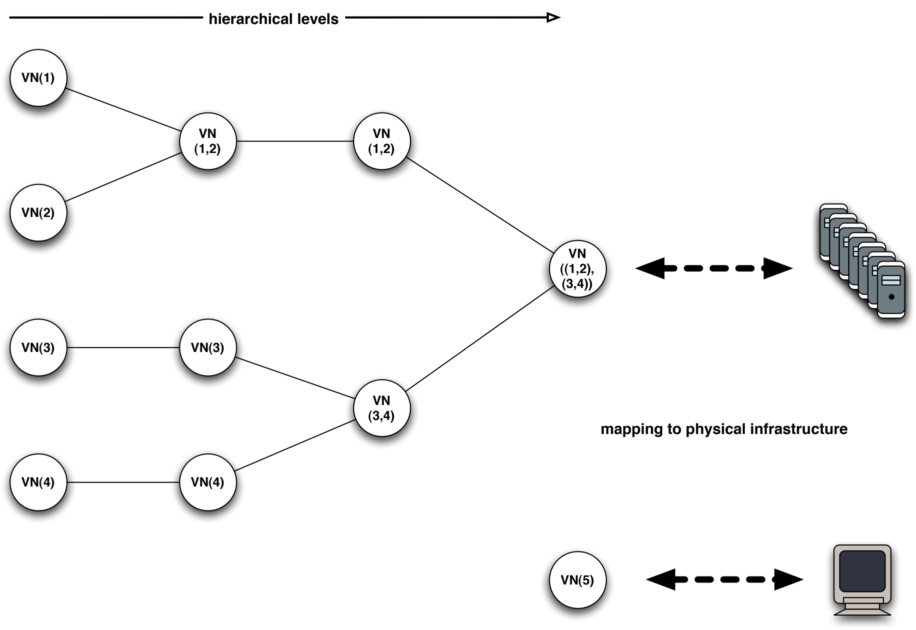


Figure 3.3: *Composition of virtual nodes in a hierarchical architecture*

colocated on the same dedicated cluster or subnetwork of desktop machines for instance. Component F may be a pre-packaged component which exposes VN(1,2), so that it may be integrated within the whole system and benefit from the virtual node composition.

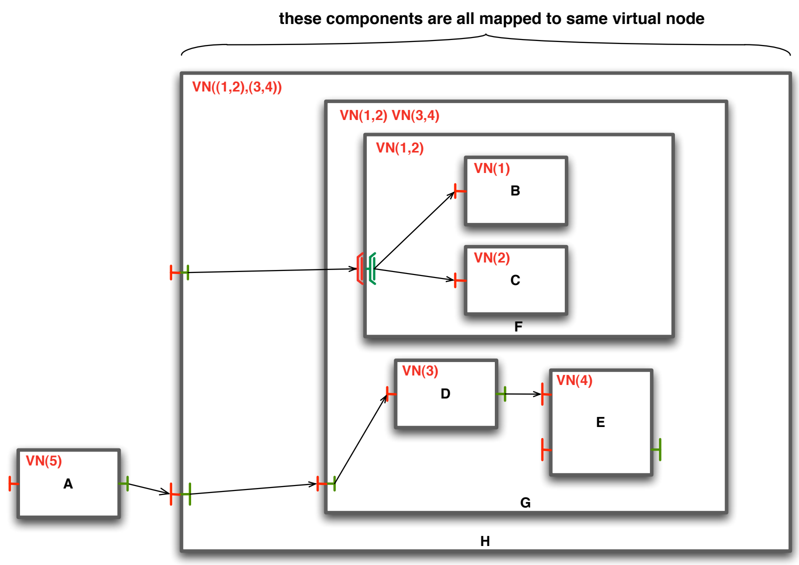


Figure 3.4: *Composition of virtual nodes and components*

3.3.3.4 Collective operations for deployment

The deployment of a Fractal component system involves: the creation of remote components, the assembly and the connections between the components. Grids are environments where the number of components may be high (hundreds, thousands), and therefore it is beneficial to provide parallelization primitives for the deployment process, so that deployment time is optimal.

For this purpose, we use a combination of the ProActive Typed Groups mechanism and of extensions to the Fractal API. These extensions were used in the Jem3D application described in section 6.2, in order to deploy in parallel a large number of components on a Grid.

Creation of components We added the following primitives to the component factory (see section 2.2.7) of ProActive/Fractal:

```
List<Component> newFcInstanceAsList(
        Type type, ControllerDescription controllerDesc,
        ContentDescription[] contentDesc, VirtualNode virtualNode)
         throws InstantiationException;

List<Component> newFcInstanceAsList(
        Type type, ControllerDescription controllerDesc,
        ContentDescription contentDesc, Node[] nodes)
         throws InstantiationException;

List<Component> newFcInstanceAsList(
        Type type, ControllerDescription controllerDesc,
        ContentDescription contentDesc, VirtualNode virtualNode)
         throws InstantiationException;

// contentDesc is a table as content may vary for each node
List<Component> newFcInstanceAsList(
        Type type, ControllerDescription controllerDesc,
        ContentDescription[] contentDesc, Node[] nodes)
         throws InstantiationException;
```

These methods explicitly return lists of components, which can be created in parallel, using multithreading. If a virtual node is given as the location parameter, these components are instantiated on the infrastructure mapped to the virtual node, with one instance of component per underlying node. If a list of nodes is given as the location parameter, one instance of component is created on each node.

Assembly of components The assembly process consists of putting components inside composite components; it is realized by invoking the content controller. For parallelizing this process, we extended Fractal's `ContentController` into `ProActiveContentController`, which adds the following operations:

```
public void addFcSubComponent (List<Component> subComponents)
         throws ContentControllerExceptionListException;

public void removeFcSubComponent (List<Component> subComponents)
         throws ContentControllerExceptionListException;
```

When invoking these methods, assembly operations are performed in parallel using multithreading.

Binding of components The binding process uses binding controllers. So far, we did not extend the binding controller, as we do not intend to parallelize the binding process of a given component to several others, but rather to parallelize the binding process of several components of the same type to a given component. This is done using the standard group mechanism: the list of components created in the creation operation may be seen as a *typed group* using the following operation:

```
Component typedGroupOfComponents = (Component)((ProxyForComponentGroup)
    myListOfComponents).getGroupByType();
```

Then the binding process uses the standard mechanism of Typed Groups:

```
Fractal.getBindingController(typedGroupOfComponents).bindFc("i1", serverInterface);
```

The result of this operation is a binding from the interface i1 of all components of the group, to the serverInterface given as a parameter. The binding process is parallelized through the standard multithreading mechanism of the ProActive Groups.

ADL integration Some of these collective operations for deployment are already integrated in the deployment process specified by ADLs, and we intend to integrate all of them in a near future; this task is part of the ADL pattern-based deployment proposal in section 7.2.

3.3.4 Dynamic configuration

Arbitrary changes in configurations of components and component assemblies are addressed in the Fractal specification by defining factories and configuration controllers for attributes, life-cycle, bindings and content, which can all be accessed at runtime.

In the Java API of Fractal, most methods of the standard control interfaces used for dynamic reconfigurations declare throwing exceptions, therefore in our model which relies on active objects, calls on these methods generate synchronous invocations.

Unfortunately, dynamic reconfigurations with synchronous invocations is a source of locking or deadlock situations. In this section, we focus on the life-cycle of components: we identify the possible locking situations and propose solutions to avoid these problems.

3.3.4.1 Locking issues in synchronous systems

To our knowledge, in the other implementations of Fractal, invocations are synchronous. Unfortunately, even in simple cases, reconfigurations are lock- or even deadlock-prone. An elementary example is given in figure 3.5. In this example, two components are to be stopped, as commonly required when reconfiguring a system. We can observe that for stopping the two components A and B (for instance because they are part of the same composite), the ordering of request is fundamental, as shown in the following scenario:

1. B receives a stop request: it stops.
2. A calls the foo method on B, either because A defines its own activity, or because it is processing an invocation from a tier component.
3. A receives a stop request.

The Fractal specification intentionally *does not specify the behavior of a functional invocation on a stopped component*: this is left to specify in either a model extension or the implementation. If the invocation is suspended - as in the Julia reference implementation - the presented scenario is a locking situation. As shown in figure 3.5, the `foo()` invocation cannot proceed, therefore A cannot be stopped, as it is currently calling B.

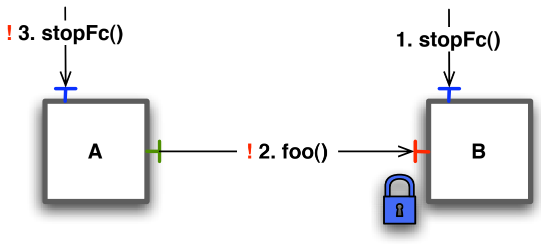


Figure 3.5: *A simple locking situation when stopping two components*

In a framework such as DREAM, based on the Julia implementation, component systems may be multithreaded as components may define their own activity (thus A may proactively call `foo()` on B after B is stopped). Locking problems are addressed when stopping subsystems which contain active components, as in [LAU 04], by defining stopping policies in the configuration of the component system. These policies specify the order in which components must be stopped: either with a programmed algorithm, with a sequential list, or by implementing custom life-cycle controllers. Unfortunately, this solution appears quite complex, especially in systems with many active components. Moreover, specifying the stopping policy in a configuration file is not valid anymore when the system is reconfigured.

In the component model we propose, each component is active, therefore the locking problem identified in DREAM in the specific case of subsystems which may contain an active component is generalized in our model.

3.3.4.2 Locking issues with active components

In our model, the locking problem described above remains. Because incoming invocations are processed sequentially and non-preemptively, A needs to finish the current service that generated the `foo()` invocation before processing the `stopFc()` invocation, but it cannot finish the current service as B is stopped.

We propose to avoid this situation by:

- managing reconfigurations from a unique place, so that reconfiguration calls do not interfere, and
- properly scheduling the stop operations from the component which manages the reconfiguration; this must be done programmatically in the reconfiguration entity, and requires knowledge of the components involved in the reconfiguration.

Recursive stopping issues Because composite components are active components, recursive assemblies of components may lead to deadlocking situations. For instance, if stopping is performed recursively, we cumulate the potential locking problem raised above with the problem of pending communications from the inside towards the outside of the composite, as shown in figure 3.6.

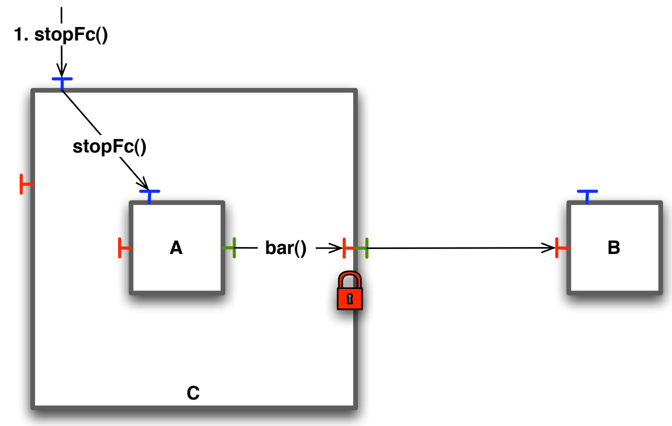


Figure 3.6: *Recursive stop operation leading to a deadlock*

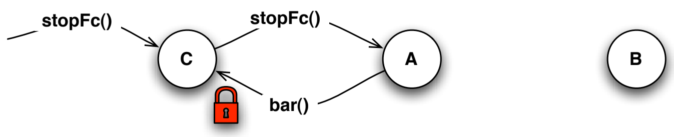


Figure 3.7: *Recursive stop operation leading to a deadlock: involved active objects*

A different view representing the active objects involved is provided in figure 3.7.

Indeed, if the stop operation is recursive, C starts processing the stop request, and sends a stop request to the components it contains, in this case A. However A is still busy processing some pending requests, and one of them requires a communication with B, which goes through C. In our model, as C is an active object, it enqueues the request from A. As requests are sequentially processed, C cannot serve the request from A before finishing processing the stop request. The stop request involves a synchronous request to stop A, which cannot be processed until A terminates processing its current service. This is a deadlock situation.

Other problematic situations occur when relationships between components create cycles. It is generally admitted, as pointed out in [HER 00], that software systems should avoid circularities and be designed as Directed Acyclic Graphs (DAGs). Hierarchical composition and injection dependencies however hinder the enforcement of this pattern.

The Fractal specification does not require composite components to have a recursive stopping algorithm. This is left for the implementation to specify. In Julia for instance, the algorithm is not recursive, because of possibly shared components.

How to overcome the deadlock problem? One solution, although arguably not a very efficient one, is to avoid using recursive stops, and instead, individually stop each component in the correct order. Another one would be, as in our model there are no shared components so far, to modify the life-cycle control API, so that stop invocations are fully asynchronous, in other words, that the stop method does not declare throwing any exception. This is however a modification with implications on the Fractal API itself, and should probably be generalized: in an asynchronous system, is it possible to have an asynchronous API for controlling components, in which methods would never declare throwing any exception?

Another approach to investigate, along with the asynchronous API, is the use of domain-specific languages to define reconfigurations. In [DAV 06], the authors propose *FScript*, a language which allows the definition of structural reconfigurations applicable to a running application in a consistent manner.

3.4 Conclusion

In this chapter we defined a component model applicable for Grid computing, based on the Fractal component model and the active objects model.

We motivated the choice of Fractal as a basis for our component model by considering its properties as a component model, and by evaluating its adequacy in the context of Grid computing. We identified three main areas we contributed to in order to suit Grid computing requirements:

- the definition of components as independent distributed entities of variable granularities,
- the specification of collective interfaces,
- the specification of a deployment model.

After a presentation of the specificities related to the underlying active object model, we presented the proposed deployment model and the facilities provided for the control of the distribution and the parallelization of deployment.

We identified issues related to the combination of hierarchical systems and synchronous API invocations with a model based on active components, including active composite components. We proposed general solutions to avoid these issues. We pointed out that these are complex distributed synchronization problems, and raised the question of defining an asynchronous control API.

Regarding the requirements of Grid computing as laid out at the beginning of this chapter, the component model we propose handles or facilitates distribution, deployment, complexity and dynamicity. Requirements of heterogeneity, legacy software distribution, multiple administrative domains with security constraints, heterogeneity, legacy software are left to the underlying middleware; in our implementation we use the ProActive middleware for these purposes.

Managing collective interactions between the components is our third main contribution and it is detailed in the next chapter.

Chapter 4

Collective Interfaces

Simple things should be simple and
complex things should be possible.
Unless simple things are simple,
complex things are impossible.

Alan Kay

Collective communications provide facilities to manage interactions between distributed entities. Computational patterns extensively use collective communications, such as in master-slave, divide-and-conquer or SPMD models. In this work we are interested in the parallelism and synchronization properties offered by collective communications, and we want to express collective behaviors in the definition of the components. Publish-subscribe or event-reaction models, which are commonly implemented by using message-oriented middleware (such as JMS in Java), are out of scope because they use intermediate components.

In order to simplify the programming model and the assembly of component systems, we want to integrate collective communications in the specification of the component model.

In this section, we describe our proposal for adding collective interfaces to the proposed component model based on Fractal, by introducing multicast and gathercast interfaces. Moreover, we demonstrate how these collective interfaces can be used as a basis for facilitating the design of SPMD programs, and how they can model the MxN redistribution problem.

4.1 Motivations

4.1.1 On the current cardinalities of component interfaces in the Fractal model

In the type system of the Fractal model, a component type is just a set of interface types. The type of the component is given by the types of its *external* interfaces, and a component of type T must have as many external interfaces as described in the type T. If the component exposes its content, in other words if it is a composite component, then the component must also have - at most - as many internal functional interfaces as described in T. This implies that each internal functional interface has a complementary external interface of the same name, signature, contingency and cardinality, of opposite role.

This chapter focuses on the cardinality of the interfaces. In the current Fractal specification, the cardinality is a property of an interface type that indicates how many interfaces of this type a given component may have. Currently, the Fractal model defines two kinds of cardinalities for interfaces: *singleton* and *collection*.

- Interfaces of cardinality *singleton* are unique and exist at runtime. Single interfaces can have a server role (Fig. 4.1.a), or a client role (Fig. 4.1.b). Fig 4.1.c and Fig 4.1.d show respectively a singleton server interface and a singleton client interface for primitive components. In the case of composite components, there is a complementary internal interface of opposite role associated with each external interface: Fig 4.1.e shows a single client interface along with its complementary internal single server interface, while Fig 4.1.f shows a singleton server interface along with its complementary internal singleton client interface.

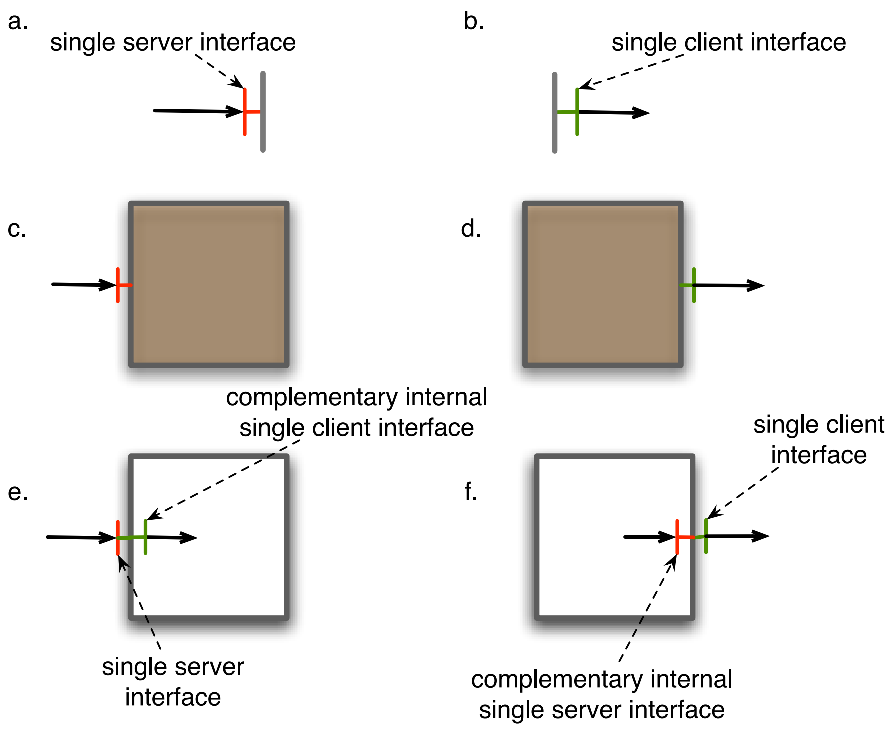


Figure 4.1: *Single interfaces for primitive and composite components*

- Interfaces of cardinality *collection* represent collections (i.e. an arbitrary number of lazily created interfaces) of interfaces of the same type, with a common prefixed name. One can define collection server interfaces (Fig. 4.2.a), or collection client interfaces (Fig. 4.2.b). A collection interface does not exist at runtime, only the lazily created interfaces that are part of the collection are accessible at runtime (see Fig 4.2). In the case of composite components, there is a complementary internal interface associated with each lazily created interface (Fig. 4.2.e and 4.2.f), which is not the case for primitive components (Fig. 4.2.c and 4.2.d). In order to perform an invocation on all of the interfaces connected to a collection interface, it is necessary first to get a reference on each of the interfaces, and second perform the invocation on each of these references. Collection interfaces only provide one-to-one bindings. In this current work, we intend to provide a mechanism for performing collective invocations without having to refer to each of the interfaces successively.

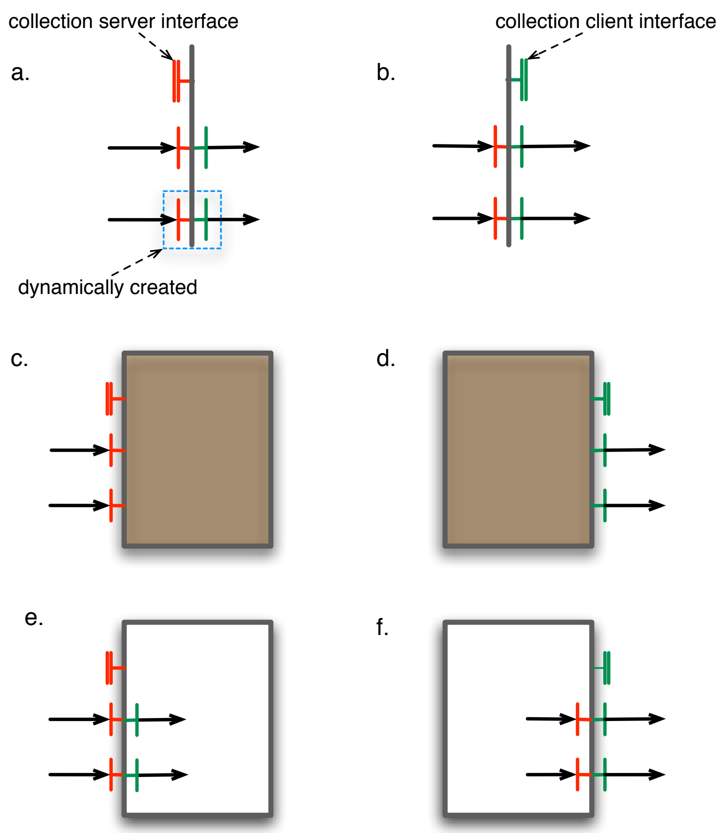


Figure 4.2: *Collection interfaces for primitive and composite components*

4.1.2 Rationale for collective interfaces

As we can see, the current cardinalities of Fractal interfaces allow only one-to-one communications. It is possible though to introduce *binding components*, which act as brokers and may handle different communication paradigms. Using these intermediate binding components, it is therefore possible to achieve one-to-n, n-to-one or n-to-n communications between components. It is not possible however for an interface to express a collective behavior: explicit binding components are needed in this case.

In response to some of the specific requirements and conditions of Grid computing, namely complexity and performance, we propose the addition of new cardinalities in the specification of Fractal interfaces, namely *multicast* and *gathercast*.

Multicast and *gathercast* interfaces give the possibility to *manage a group of interfaces as a single entity* (which is not the case with a collection interface, where the user can only manipulate individual members of the collection), and they *expose* the collective nature of a given interface. Specific semantics for multiway invocations can be configured, providing users with flexible communications to or from gathercast and multicast interfaces (see section 2.4). Moreover, the specification of a component includes communication semantics when they are specified at the level of interfaces. Finally, avoiding the use of explicit intermediate binding components simplifies the programming model and type compatibility is automatically verified.

The role and use of multicast and gathercast interfaces are complementary. Multicast interfaces are used for parallel invocations and data distribution, whereas gathercast interfaces are used for synchronization, gathering or redispatching purposes.

This proposal is intended to be both general and clear: general enough so that it is not tied to any implementation, and clear enough so that there is no ambiguity and that all implementations work the same way.

4.2 Proposal for collective interfaces

We propose to integrate the notion of collective interfaces into the component model, in order to expose a specific collective behavior at the level of an interface.

Collective interfaces correspond to new kinds of cardinalities[1] interfaces: *multicast* and *gathercast* (with gather-multicast as a perspective). Each of these cardinalities provides facilities for collective communications, and their behavior is customizable.

In this section, we present multicast and gathercast interfaces. For each one, we detail their specificities, in terms of data distribution and method invocation, and discuss their specific properties.

Preliminary notes:

- The examples provided in this section use the Java language, but the proposal is not tied to this language.
- In the sequel, we use the term List to mean ordered set of elements of the same type (modulo sub-typing). This notion is not necessarily linked to the type `List` in the chosen implementation language; it can be implemented via lists, collections, arrays, typed groups, etc. To be more precise, we use `List<A>` to mean "list of elements of type `A`".

4.2.1 Multicast interfaces

Multicast interfaces provide abstractions for one-to-many communications. After a general definition of multicast interfaces, we clarify the distinction between multicast server and client interfaces, then we describe the mechanisms of the invocations, by showing the different types of invocations, the management of the results and the distribution of the parameters. An illustration is eventually provided through some programming examples.

4.2.1.1 Definitions

The following definitions characterize external interfaces (which define the type of a component).

Definition 4.2.1 *A multicast interface transforms a single invocation into a list of invocations.*

[1]The term *cardinality* is extended from the original Fractal specification: in this proposal, the cardinality of an interface also designates 1-to-n and n-to-1 connections with specific semantics

When a single invocation is transformed into a list of invocations, these invocations are forwarded to a set of connected server interfaces. A multicast interface is unique and it exists at runtime (it is not lazily created). The semantics of the propagation of the invocation and of the distribution of the invocation parameters are customizable, and the result of an invocation on a multicast interface - if there is a result - is always a *list of results*. Invocations forwarded to the connected server interfaces may occur in parallel, which is one of the main reasons for defining this kind of interface: it enables *parallel invocations*.

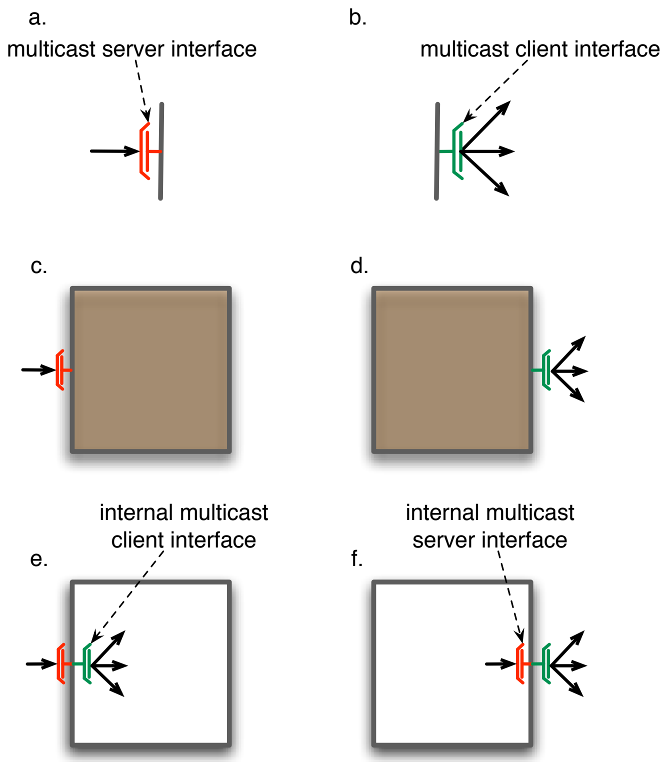

a.
multicast server interface
b.
multicast client interface
c.
d.
internal multicast client interface
e.
internal multicast server interface
f.

Figure 4.3: *Multicast interfaces for primitive and composite components*

4.2.1.2 Multicast client interfaces vs multicast server interfaces

On one hand, a multicast client interface distributes invocations to connected server interfaces. On the other hand, a multicast server interface explicitly exposes a multicast behavior (notably the fact that a result is actually a list of results), and forwards a single invocation either to a complementary multicast client interface in the case of a composite component (Fig. 4.3.e), or to a contractually defined implementation code in the case of a primitive component (Fig. 4.3.c).

4.2.1.3 Signatures of methods

Multicast interfaces may distribute data in different manners, and may receive results from several connected interfaces. This has a consequence on the signature of the methods of multicast interfaces.

Return type For each method invoked and returning a result of type `T`, a multicast invocation returns an indexed aggregation of the results: a `list`, and if the language allows it, a `list of T`.
There is a *type conversion*, from return type `T` in a method of the server interface, to return type `list of T` in the corresponding method of the multicast interface (as represented in figure 4.4.a). The framework must transparently handles the type conversion between return types, which simply corresponds to an aggregation of elements of type `T` into a structure of type list of `T`.

To sum-it up, consider the signature of a server interface:

```java
public interface I {
   public void foo();
   public A bar();
}
```

A multicast interface can be connected to the server interface with the above signature only if the signature of this multicast interface is the following:

```java
public interface J {
   public void foo();
   public List<A> bar();
}
```

Parameters types Parameters may be distributed in various manners, but signatures are impacted in only two possible ways. If a given parameter is broadcasted, the signature for this parameter must be compatible on both sides (the parameter on the client side must be of the same type, or a subtype of the parameter on the server side). However if a given parameter is to be scattered, this implies it is fed to the multicast interface as a `list of T` and the server interfaces receive parameters of type T, as represented in figure 4.4.b). Impact of distribution on method signatures is further detailed in this chapter, but to introduce the concept, consider the signature of a client multicast interface:

```java
public interface J {
   public void foo(List<A> la); // parameter to be scattered
   public void bar(List<B> lb); // parameter to be broadcasted
}
```

This multicast interface with the above signature and properties (concerning parameters distribution) may be connected to a server interface only if the signature of this server interface is the following (modulo subtyping):

```java
public interface I {
   public void foo(A a);
   public void bar(List<B> lb);
}
```

An illustration of the compatibility of signatures for component systems using a multicast interface is presented in figure 4.4: *a* focuses on the return types, whereas *b* illustrates a multicast interface with a scatter distribution mode.

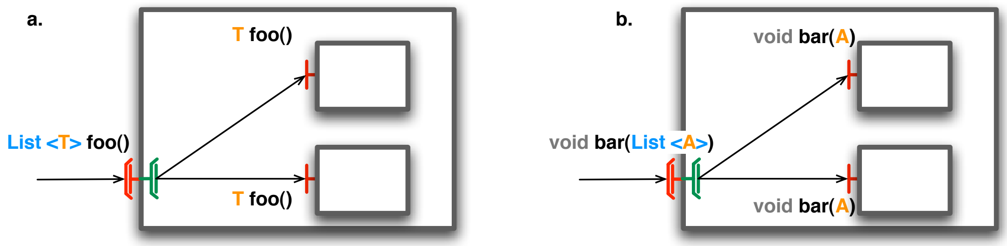


Figure 4.4: *Component systems using multicast interfaces*

4.2.1.4 Invocation forwarding

A multicast invocation leads to the invocation of a method offered by one or several connected server interfaces. The communications may be synchronous or asynchronous, and can be performed in parallel. We consider a multicast invocation as a one-to-many communications, i.e. from a caller interface to all connected interfaces. Other dispatch modes, such as one-to-some or one-to-one, similar to ICENI's dispatch tee (section 2.4.3.3), are not considered in this proposal, but they are envisaged in section 7.4.

When performing an invocation to or from a multicast interface, the transfer of the invocation - from client to connected server interfaces - occurs according to a user-defined policy, and lets the possibility for users to define *custom* communication modes[2] . Therefore, the communication scheme policy must be part of the definition of the multicast interface. It is convenient to specify it in the signature of the interface.

Dynamic dispatch We consider the common situation where:

1. a multicast interface is bound to several server interfaces,
2. the invocations can be distributed regardless of which component will process the invocation,
3. inter-component communications are asynchronous.

This situation corresponds for example to the case of stateless server components (more generally when the processing of invocations is independent from previous invocations), or to the case of embarrassingly parallel computations. In order to improve the global performance of a computation in this configuration, it is possible to use techniques such as *dynamic dispatch* of invocations to connected server components. The idea is to send more invocations to faster server components than to slower components. The consequence is that the overall computation time is reduced because the invocations (which are the jobs to compute) are distributed more wisely. This principle is notably proposed in [BAD 05a].

4.2.1.5 Distribution of invocation parameters

If some of the parameters are defined as lists of values, these values can be distributed in various ways through method invocations to the server interfaces connected to a multicast interface. The default behavior - *broadcast* - is to send the same parameters to

[2]The behavior of the interface may be configured in several manners; in the Java language, one of them is through annotations on the methods or interfaces.

each of the connected server interfaces. In the case some parameters are lists of values, copies of the lists are sent to each receiver[3]. However, similar to what SPMD programming offers, it may be adequate to strip some of the parameters so that the bound components work on different data. In MPI for instance, this can be explicitly specified by stripping a data buffer and using the `scatter` primitive.

Multicast interfaces provide the possibility to automatically strip and distribute parameters to bound components. We now discuss the possible configurations, then we explain the technique we use to specify these configurations.

Distribution policies What are the possible distributions of the parameters of the invocations which are generated and forwarded from a multicast interface? Many configurations are possible, depending on the number of parameters that are lists and the number of members of these lists. The applicability of a redistribution depends on the types of the parameters of the methods of the interface. Table 4.1 indicates under which conditions a redistribution can be applied. It shows that the types of the parameters (thus the signature of the interfaces) may be different between the server interface and the multicast interface, but only in the conditions presented here. It also shows that scattering of a given parameter may only occur from a list of elements of type `T` to elements of type `T`. Note that the table only considers one parameter of one method of the interface, and that all parameters of all methods of the involved interfaces must verify the applicability conditions. The cases not marked as valid are also cases where the connection between interfaces is simply not possible (for this kind of binding), because at least the types of two parameters are incompatible. When the parameters are of the same type in the multicast interface and in the connected server interface, then the considered parameter is copied (broadcast).

	type of *ith* parameter of a method in server interface	
type of *ith* parameter of a method in multicast interface	**T**	**List<T>**
T	✓ broadcast	-
List<T>	✓ scatter	✓ broadcast
A	-	-

Table 4.1: *Applicability of a distribution function depending on the type of the i^{th} parameter of method of the multicast interface vs. the type of the corresponding parameter in the corresponding method of its connected server interfaces. A and T are incompatible types.*

Once identified adequate situations for the distribution of invocation parameters, one can define a distribution function. Modulo reordering of the parameters, we can consider that the signature of a method of the multicast interface is as follows:

- $List < A > foo(T_1, ...T_m, list < T'_1 >, ...list < T'_l >)$ *on the multicast interface*,
- $A\ foo(T_1, ...T_m, T'_1, ..., T'_n, list < T'_{n+1} > ..., list < T'_l >)$ on the connected server interface

[3]This is a good place for optimization by preserializing data to be sent.

where:

- foo is the method which is invoked and where $n \leq l$. If $l = n$, then the signature becomes $foo(T_1, ...T_m, T'_1, ..., T'_n)$ on the server interface side,
- n is the number of parameters that will be scattered,
- m is m the number of non-list parameters,
- $l - n$ the number of list parameters that are broadcasted as full lists, l being the number of lists on the multicast interface.

The order of the parameters does not correspond to the actual signature of the methods and is just a way to simplify the representation. Depending on the configuration of the interface, some of the lists have their elements broadcast, and others have their elements scattered.

In other words, method calls on a multicast interface of the form $foo(a_1, ...a_m, L_1, ...L_l)$ are forwarded as several method calls of the form $foo(a_1, ...a_m, a'_1, ...a'_n, L_{n+1}, ...L_l)$ where a_i and $a^{'}_i$ are not (necessarily) lists, and L_i are lists.

Parameters $a_1, ...a_m$ that are not lists of values in methods of the multicast interface are systematically copied in each invocation which is generated (broadcast). Parameters $L_{n+1}, ...L'_l$ that are lists but correspond to lists on the connected server interfaces are also systematically copied in each invocation (broadcast). Parameters $L_1, ...L_l$ that are lists of values of type T and correspond to arguments of type T on the connected server interfaces are distributed (scattered) according to the following transformation, which provides the different combinations for the parameters: $f \subseteq \mathbb{N}^n$ or more precisely, $f \subseteq [1..k_1] \times [1..k_2] ... \times [1..k_n]$ where f is the (multi)set of the combinations of parameters, n is the number of parameters of the invoked method which are lists of values, and $k_i, 1 \leq i \leq n$ the number of values for each list parameter.

Three cases are of particular interest to us in the context of parallel computing. The first one corresponds to $\forall i, j \in [1..n], card(L_i) = card(L_j)$: all list parameters have the same number of values. A straightforward redistribution gives for the i^{th} invocation: $foo(a_1, ...a_m, L_{1_i}, ...L_{n_i})$. In this case, n invocations are generated and forwarded.

The second notable case is the one that explores all combinations: $f = [1..k_1] \times [1..k_2] ... \times [1..k_n]$, which results in invocations like $foo(a_1, ...a_m, L_{1_{i_1}}, ..., L_{n_{i_n}})$ where $i_j \in [1..card(L_j])$. It is a cartesian product, and $\prod_{i=1}^{n} card(L_i)$ gives the number of invocations that are generated and forwarded. There may be more invocations than the number of connected server interfaces.

The third interesting case is the one where *all* the parameters are simply copied and sent to each receiver. It corresponds to the case where $n = 0$ in the above description of the signature of the method in the server interface. Each receiver receives the same parameters; this corresponds to the *broadcast* mode. The i^{th} invocation therefore corresponds to: $foo(a_1, ...a_m, L_1, ...L_l)$: the parameters are identical in the method of the multicast interface and in the one of the bound server interfaces, and the number of invocations is the number of bound server interfaces.

Specification of distribution policies Each eligible parameter (according to table 4.1) may be subject to a distribution operation, therefore it must be possible to specify a distribution policy for each eligible parameter of each method of a multicast interface. The distribution policy may be defined globally for all the parameters of all the methods of a given interface, or more specifically at the level of the methods, in which case the policy is applied for all the parameters of this method. For these globally specified policies, *the number of elements must be the same in each scattered list.*

The distribution policy may also be specified individually for each eligible parameter. In that case of course the different distribution policies among the parameters must be compatible: *the resulting number of dispatched entities should be the same for each stripped parameter*. This must be verified at runtime and depends on the number of connected server interfaces, on the number of elements for each scattered list, and on the distribution policies for each parameter. Contrarily to globally specified policies, this does not imply that the number of elements must be the same in each scattered list.

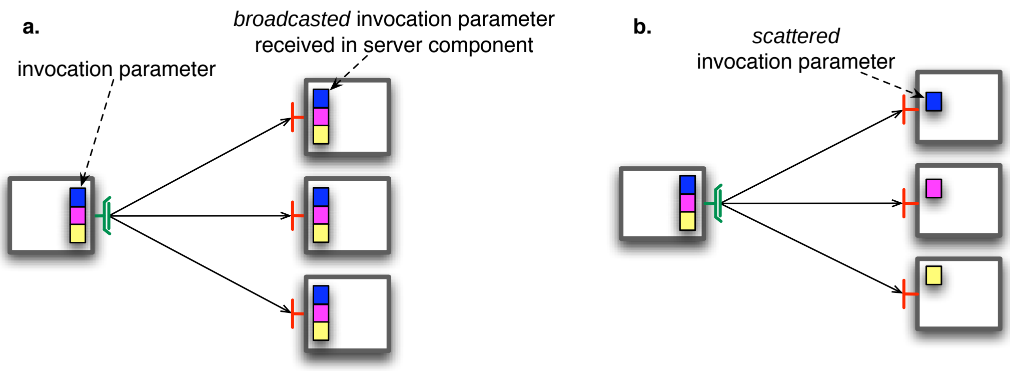

a.
invocation parameter
broadcasted invocation parameter received in server component
b.
scattered invocation parameter

Figure 4.5: *Broadcast and scatter of invocations parameters*

4.2.1.6 Distribution of invocations

Once the distribution of the parameters is determined, the invocations that will be forwarded are known. A new question arises: how are these invocations dispatched onto the connected server interfaces? This is determined by another function d, which, knowing s the number of server interfaces bound to the multicast interface, and g the number of invocations that have been generated (from the previous function f), is defined as follows: $d : [1, g] \longrightarrow [1, s]$.

When $s = g$, d can correspond to the identity function I: the generated invocation of index i is sent to the bound interface of index i.

When $s < g$, invocations may be distributed in a round-robin fashion among the available links to server interfaces. Another possibility is the dynamic dispatch of invocations, optimizing the distribution of tasks based on the responsiveness of each server component.

When $s > g$, some of the bound interfaces do not receive any invocation, but it is possible to keep an internal state for the invocations, so that those interfaces will be selected first when dispatching new invocations.

4.2.1.7 Management of results

An invocation on a multicast interface usually generates several forwarded invocations. If the invoked method returns a value, then the invocation on the multicast interface returns an ordered collection of result values: a *list*, as mentioned in 4.2.1.3. This implies that, for the multicast interface, the signature of the invoked method has to explicitly specify *list* as a return type. This also implies that each method of the interface returns either nothing, or a list.

In object-oriented languages, the type of the elements of the list may be specified using generics programming such as templates in C++ or Generics in Java 1.5.

It is possible to use typed lists, such as for examples typed groups in the ProActive library [BAD 05a].

Some implementations of these proposals may use arrays, in which case the term list refers to a one-dimensional array.

When manipulating multicast interfaces, users know that they have to expect *lists of results* from invocations on these interfaces, and therefore they must handle these lists of results accordingly.

We envisage, in a future extension of this proposal, to propose a reduction mechanism that would return one (or several) reduced value(s) instead of systematically returning a list of aggregated values. Reduction would therefore have an incidence on the signature of the multicast method, which may not return a list, but rather a single reduced element.

4.2.1.8 Programming example

A multicast client interface of signature `MyInterface`, named “myMulticastInterface", that is defined in the type of the component, exists at runtime as a collective interface named “myMulticastInterface", of cardinality “multicast”. The type of this interface is defined as follows:

```
InterfaceType itfType = typeFactory.createFcItfType(
      "myMulticastInterface",
      MyInterface.class.getName(),
      TypeFactory.CLIENT,
      TypeFactory.MANDATORY,
      TypeFactory.MULTICAST
   )
```

The management policy for the collective interface is specified at the construction of the component, in an implementation specific way. One way to specify the management policy is by using attribute-oriented programming, annotating program elements (interfaces, methods or parameters in this case) with metadata information. In our implementation, described in section 5.3, the distribution policy is specified in the Java interface, using Java annotations, a mechanism for adding metadata, which was introduced in the Java language in version 5.

In this example, we would have the `MyInterface` interface defined as:

```
@ClassDispatchMetadata(
   mode=@ParamDispatchMetadata(mode=ParamDispatchMode.ONE_TO_ONE))
public interface MyInterface {

 public List<A> foo(List<B>, C);

}
```

The dispatch mode is defined at the level of the declaration of the interface and therefore applied to *all eligible parameters of all methods* of the interface. The distribution mode is *one-to-one*, meaning that `List<B>` must contain as many elements as connected server interfaces. B elements are scattered and dispatched to the connected server interfaces, and a copy of C is received by each connected server interface.

The different modes and configurations available in our implementation are described in details in section 5.3.

Multicast interfaces are bound to interfaces of compatible types using the standard binding mechanism:

```
bindingController.bindFc(
   "myMulticastInterface",
   serverItf1
   );
```

Provided serverItf1 is of a compatible type, this adds the serverItf1 interface to the group of interfaces managed by the multicast interface.

4.2.1.9 Application to the master-worker pattern

The master worker pattern is suitable for embarrassingly parallel problems, commonly used for many of scientific areas, as demonstrated by the variety of projects (from genomics to the search of extraterrestrial intelligence) using the BOINC infrastructure [AND 04], or the network enabled server environments (see section 2.1.2.4). This pattern is trivially designed using multicast interfaces, and custom distribution patterns may be specified at ease. In [BOU 06], the authors propose to design master-worker patterns using intermediate components for handling the distribution. In our proposal, the distribution is handled at the level of the component interface, in the control membrane, and therefore *does not require any intermediate component*. Moreover, we clearly define the semantics of the collective communications, so that the necessary 1-to-n communications are properly specified. The authors of the aforementioned paper also propose to specify and parameterize the master-worker pattern in the ADL. We concur with the idea of taking advantage of ADLs to specify common assembly patterns that simplify design tasks, and one of our current investigation topics is to offer facilities for specifying distributed component assemblies and communication patterns in ADLs, as described in section 7.2.

4.2.2 Gathercast interfaces

Gathercast interfaces provide abstractions for many-to-one communications. In this section, after a general definition of gathercast interfaces, we clarify the distinction between multicast server and client interfaces, we consider the binding mechanisms for this type of interfaces, and we describe the synchronization and data gathering capabilities. An illustration is eventually provided through some programming examples.

4.2.2.1 Definition

The following definition characterizes external interfaces.

Definition 4.2.2 *A gathercast interface transforms a list of invocations into a single invocation.*

The gathercast term originates from previous works on IP networks [BAD 00], where the authors proposed a mechanism for aggregating IP packets. Our current study somehow proceeds from these previous works, though it does not only address data aggregation, but also process coordination. Indeed, a gathercast interface can coordinate incoming invocations before continuing the invocation flow: it can define synchronization barriers and gather incoming data. Invocation return values are also redistributed to the invoking components.

A gathercast interface is unique and it exists at runtime. Synchronization barriers and gathering operations are customizable, as well as redistribution policies for invocations return values.

4.2.2.2 Gathercast client interfaces vs gathercast server interfaces

The comparison between gathercast client and gathercast server interfaces is similar to the comparison between multicast client and multicast server interfaces. Note that we always consider external interfaces, unless explicitly mentioned otherwise.

Gathercast server interfaces gather invocations from multiple client interfaces (figure 4.6), but client interfaces can also have a gathercast cardinality. Gathercast client interfaces transform gathercast invocations (with gathering and synchronization operations) into a single invocation which is transferred to a bound server interface, and client gathercast interfaces also explicitly expose their gathercast nature, indicating that invocations coming from this client interface contain gathered parameters: parameters of the forwarded invocation are typed lists. In primitive components, the purpose of a gathercast client interface is solely to expose the gathercast nature of this interface.

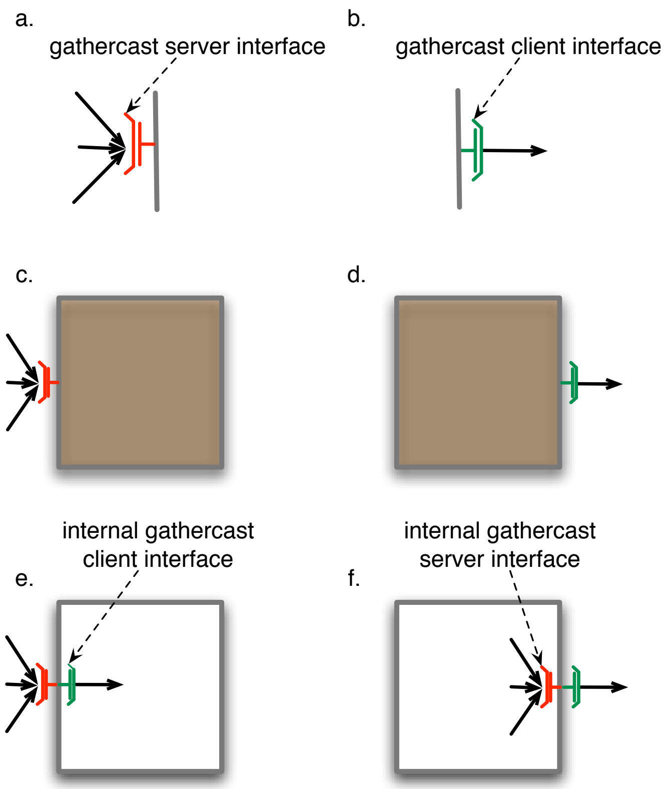

a.
gathercast server interface
b.
gathercast client interface
c.
d.
internal gathercast client interface
e.
internal gathercast server interface
f.

Figure 4.6: *Gathercast interfaces for primitive and composite components*

4.2.2.3 Bindings to gathercast interfaces

The Fractal model defines a binding as a link between two interfaces. It does not explicitly specify the link as unidirectional, though one could interpret the specification as stating that a binding is a link from an interface i1 to an interface i2, whose only goal is to provide an access to i2 from i1 in order to invoke operations on i2.

The general idea of our proposal is to manage the semantics and behavior of the collective communication at the level of the interface itself. Gathering operations require knowledge of the participants (i.e. the clients of the gathercast interface) of the collective communication. Therefore, the binding mechanism, when performing a binding to a gathercast interface, must make the gathercast interface aware of this binding. This awareness must be handled transparently by the framework, while keeping the standard Fractal binding operations: binding operations to gathercast interfaces can be detected by the framework and handled in such a way that the gathercast interface receives a reference on the interface which is being bound to it.

As a consequence, in the context of gathercast interfaces, we have to explicitly state that *bindings to gathercast interfaces are bidirectional links*, in other words: a gathercast interface is aware of which interfaces are bound to it.

4.2.2.4 Synchronization operations

Synchronization operations can be based on the data (messages) or on the processes.

Gathercast interfaces provide message-based synchronization capabilities: the message flow can be blocked on user-defined message-based conditions. *Synchronization barriers* can be set on specified invocations, for instance the gathercast interface may wait - with a possible timeout - for all its clients to perform a given invocation on it before forwarding the invocations. It is also possible to define more complex or specific message-based synchronizations, based on the content of the messages or based on temporal conditions, and it is possible to combine these different kinds of synchronizations.

4.2.2.5 Data operations

The gathercast interface aggregates parameters from method invocations, therefore the parameters of an invocation coming from a gathercast (client) interface are actually lists of parameters. The result of the invocation may be a simple result, or a list of results, in which case a redistribution of the enclosed values may occur.

Gathering of parameters When several components invoke a given method on a gathercast interface, it is possible to aggregate the arguments of the method invocations (figure 4.7). In implementations of Fractal using an object-oriented language, the aggregation structure should be a list (`java.util.List` in Java) so that elements are indexed. This list is passed to the interface bound to the gathercast client interface.

Redistribution of results The distribution of results for gathercast interfaces is symmetrical with the distribution of parameters for multicast interfaces, and the configuration of the distribution may be handled in a similar manner, provided each client interface participating to the gather operation receives a single result.

For multicast interfaces, the redistribution function only deals with parameters which are lists of parameters. Parameters which are not lists are systematically copied in each invocation. In the case of gathercast interfaces, a similar reasoning applies to results, as

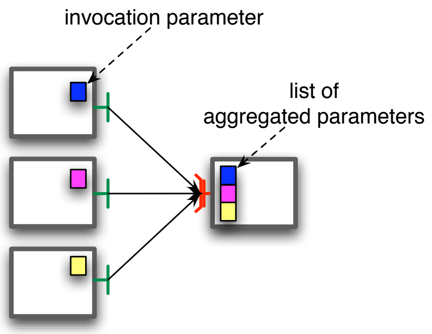


Figure 4.7: *A simple example of the aggregation of invocation parameters for a gathercast interface*

opposed to invocation parameters: it depends on the type of the result in the signature of the invoked method (table 4.2).

A redistribution function is only applicable if two conditions are fulfilled. The first one is a condition on the type of the results, and the following table considers the return types for a given method. All methods of the interfaces must verify these conditions.

		return type for a method in client interface		
		void	T	List<T>
return type for a method in gathercast interface	void	✓	–	–
	T	–	✓ (broadcast)	–
	List<T>	–	✓ (scatter)	✓ (broadcast)
	A	–	–	–

Table 4.2: *Applicability of a redistribution function depending on the return types of the methods of the gathercast interface vs. the return types of the methods of its client interfaces. A and T are incompatible types*

As we can see, the redistribution function is only applicable when the return type for the gathercast interface method is a list of elements of type compatible with the return type of the client interface method. We can also observe a symmetry between the signatures compatibility of multicast and gathercast interfaces. Indeed, scatter and broadcast distribution modes are available for the distribution of results for gathercast interfaces.

In the scatter mode, the redistribution function determines which client of the gathercast interface receives a given member of the list of results. In the broadcast mode, every client receives the same result.

The second condition for a redistribution function to be applicable depends on the number of elements in the list of results: it must be equal to the number of client interfaces participating in the gathercast communication. Indeed, if there are less elements in the list of results, that would mean some clients never receive any result, which is not acceptable. If there are more elements in the list of results, some invocations would receive two results, which is not possible. The consequences on dynamicity are discussed in the next paragraph.

An illustration of a component system which respects those conditions with a gathercast interface is given in figure 4.8

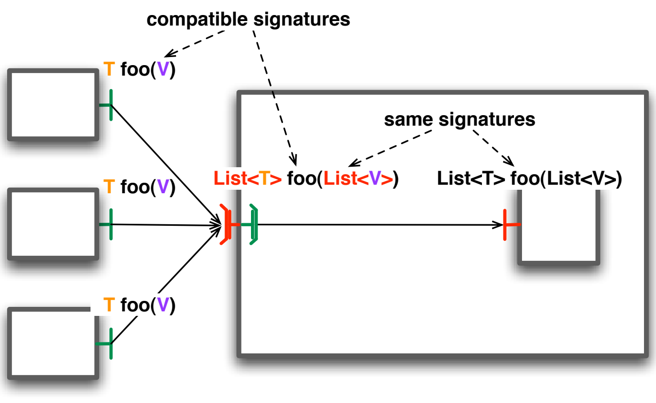


Figure 4.8: *A component system using a gathercast interface*

In conclusion, the redistribution function, applicable under the conditions stated above, is the following: f being the redistribution function, L_i the i^{th} element of the list of results, r_i the result for the client interface connected to the gathercast interface with the i^{th} index, and n the number of participating interfaces, we have: $n = card(L)$ and $\forall i, j \in [1, n], f : L(i) \longrightarrow r_j$

Redistribution functions may be configured using meta-data information, for examples Java annotations as in our implementation of multicast interfaces.

4.2.2.6 Implications on dynamicity

We note that gathercast interfaces constrain dynamicity, and this must be handled by the framework. If results of invocations are expected, and that clients connect or disconnect from the gathercast interface, the redistribution must still be coherent. There is no problem if invocations are synchronous, however, if invocations are asynchronous, then the framework must specify a policy. One possible policy is to forbid unbinding from gathercast interfaces. Other policies may be implemented by keeping a reference to unbound interfaces.

4.2.2.7 Programming example

A gathercast server interface of signature `MyInterface`, named "myGathercastInterface", that is defined in the type of the component, exists at runtime as a collective interface named "myGathercastInterface". The type of this interface is defined for instance as follows:

```
InterfaceType itfType = typeFactory.createFcItfType(
   "myMulticastInterface",
   MyInterface.class.getName(),
   TypeFactory.SERVER,
   TypeFactory.MANDATORY,
   TypeFactory.GATHERCAST
   );
```

It is possible to bind interfaces of compatible type to this gathercast interface, using the standard binding mechanism:

```
bindingController.bindFc( "myInterface1", gathercastServerItf);
```

Provided `"myInterface1"` is of a compatible type, this adds the `"myInterface1"` interface to the group of interfaces managed by the gathercast interface.

4.3 SPMD-based component programming

The objective of this section is to show how gathercast and multicast may be combined in order to realize applications in a SPMD programming style.

4.3.1 SPMD programming concepts

SPMD is a programming model for parallel computing, and arguably the most widely used pattern in High Performance Computing. It stands for Single Program Multiple Data, as each task executes the same program (which may contain conditional branching statements depending on the process ID) but works on different data. It is commonly used on clusters and parallel machines: a single program is written and loaded onto each node. Each copy of the program runs independently, on different data, and synchronization is provided through explicit messages. Each copy of the program is ranked with a unique ID. Traditionally, the language itself does not provide implicit data transmission semantics, and the communications patterns use explicit message-passing implemented as library primitives. This simplifies the task of the compiler, and encourages programmers to use algorithms that exploit locality.

The SPMD models maps easily and efficiently on distributed and parallel applications and distributed memory computing. It is the paradigm of choice for scientific parallel computing, with popular programming environments available for developers, such as MPI (Message Passing Interface) and PVM (Parallel Virtual Machines).

4.3.2 From message-based SPMD to component-based SPMD

The popularity of object-oriented programming led research teams to experiment the combination of object-oriented and SPMD programming paradigms. The MPI 2 specification was a first step, defining most of the library functions as class member functions, although at a fairly low-level. Further studies such as Object-Oriented MPI [SQU 96] provide mechanisms to build user-defined data types in accordance with the MPI specification, and communicate with them; OOMPI encapsulates the functionalities of MPI into a hierarchy of classes in order to provide a simple and intuitive interface, especially suitable for C++ bindings, but with a more general vision.

With the success of the Java language and its potential for high performance computing [BUL 03], four approaches were undertaken so as to bring the SPMD programming style to Java.

- *Wrapping of the MPI library*, and delegating invocation to an underlying MPI implementation, such as mpiJava [BAK 99] and JavaMPI [MIN 97].
- *Implementations in Java of message-passing specifications* like MPI, written in Java, such as MPJ [CAR 00], which proposes notions such as communicators or datatypes originating from MPI.

- *Group method invocations*: instead of using explicit message passing, some proposals are based on group method invocations, which allow the exchange of typed data with a group of remote processes. This suits better the object-oriented paradigm than explicit message passing, where send and receive primitives must be explicitely programmed in matching pairs. CCJ [NEL 01] is a communication library that adds MPI-like collective operations to Java, by creating groups of threads (equivalent to MPI communicators), and using RMI for communicating with these groups of threads. As a consequence, every collective operation needs a group of threads as a parameter, similarly to passing a communicator in MPI collective operations.
- *Object-oriented SPMD with active objects* (OOSPMD) is an approach proposed in [BAD 05b]. It grounds up on the concepts of active objects and typed group communications. Active objects alleviate the burden of explicitly managing synchronization issues for collective communications, and typed group communications proved adequate for easily building non-embarrassingly parallel applications. OOSPMD offers SPMD capabilities by considering groups of objects participating in a SPMD computation, by providing identification of SPMD group members, and by providing collective operations for communication and global synchronization among SPMD group members using explicit synchronization barriers.

In the context of Grid computing, the interaction between parallel codes, the deployment issues and the complexity of coupled applications led to consider more adequate programming models, such as software components, in the light of the advantages presented in 2.2. Two examples of this approach are CCA's CCAFFEINE implementation [ALL 02] and GridCCM [PÉR 03].

CCAFFEINE is an implementation of the CCA specification aiming at composing SPMD application from CCA components. It introduces the notion of SCMD (Single Component Multiple Data) components. Component communicate via ports with other components in the same address space and communicate via a process-to-process protocol (e.g. MPI) within their cohort, the SCMD set of corresponding components on all distributed processors.

As data may need to be redistributed during communications, the GridCCM model supports it as transparently as possible. The client only needs to describe how its data are locally distributed and the data is automatically redistributed accordingly to the server's preferences. Similarly to CCAFFEINE's philosophy, where all component interactions behave as if it were a serial application, a GridCCM component appears as close as possible to a sequential component. For example, a sequential component may connect itself to a parallel component without noticing it is a parallel component.

4.3.3 Component based SPMD with gathercast and multicast interfaces

We propose a novel approach for bringing the SPMD programming model into the component-based programming paradigm, using collective interfaces.

4.3.3.1 Principles

The basic requirements of the SPMD model may be listed as follows:

1. Distributed replicated software entities.

2. Asynchronous communication between these entities, either through message passing or remote method invocation.
3. Collective communications: broadcast, scatter, gather.
4. Synchronization capabilities between distributed entities.
5. Identification of entities for selective synchronization.

Our approach relies on distributed components and multicast and gathercast interfaces. The requirements of the SPMD model listed above are answered as follows:

1. ProActive/Fractal components are inherently distributed entities.
2. Components communicate through asynchronous method invocations.
3. Collective communications are provided through multicast and gathercast interfaces.
4. Synchronization is provided by gathercast interfaces, and the active object model with futures guarantees a sequential and deterministic processing of invocations.
5. Components are assembled by dependency injection; currently, ranking of component references is provided programmatically, though ADL patterns will allow an automatization of the process.

4.3.3.2 Usage and benefits

SPMD capabilities are easy to implement in our model, however the philosophy is a bit different than with standard SPMD programming. Indeed, *all SPMD features are specified outside of the functional code*, through the gathercast, multicast and collection interfaces. SPMD groups are defined by the assembly of the components, following the inversion of control pattern of Fractal binding mechanism. This brings flexibility and reuse for SPMD programs.

After the assembly is completed, the computation begins once the components are started (a "start computation" request is sent to the components, followed by a "startFc" lifecycle request). There is neither any explicit loop nor any explicit barrier: synchronization is automatic when invoking gathercast interfaces, and the computing process iterates by recursion triggered by neighbors: invoking the gathercast interface triggers the computation once the synchronization is complete, then results are sent by invoking the gathercast interface on neighbors, as represented for a typical algorithm in figure 4.9.

The assembly using gathercast interfaces provides an automatic logical synchronization (in Lamport's terms), therefore the global state of the application is always coherent at the end of the computation, even though some components may compute more rapidly than others. Figure 4.10 illustrates an SPMD system where components are linked through gathercast interfaces. The computation time for a given iteration varies among components (component 2 computes significantly faster in this example), and we can observe the effect of gathercast synchronizations: for a given component, all neighbors must send the awaited call before the computation can continue. We also represented the global logical synchronization which takes place among components, thanks to the gathercast interfaces.

In usual SPMD algorithms, the code of the application needs to explicitly handle the synchronization. The number of neighbors may vary and depends on the SPMD entity, for instance in a 2D grid like in the Jacobi example presented in 6.3.1, the entities may

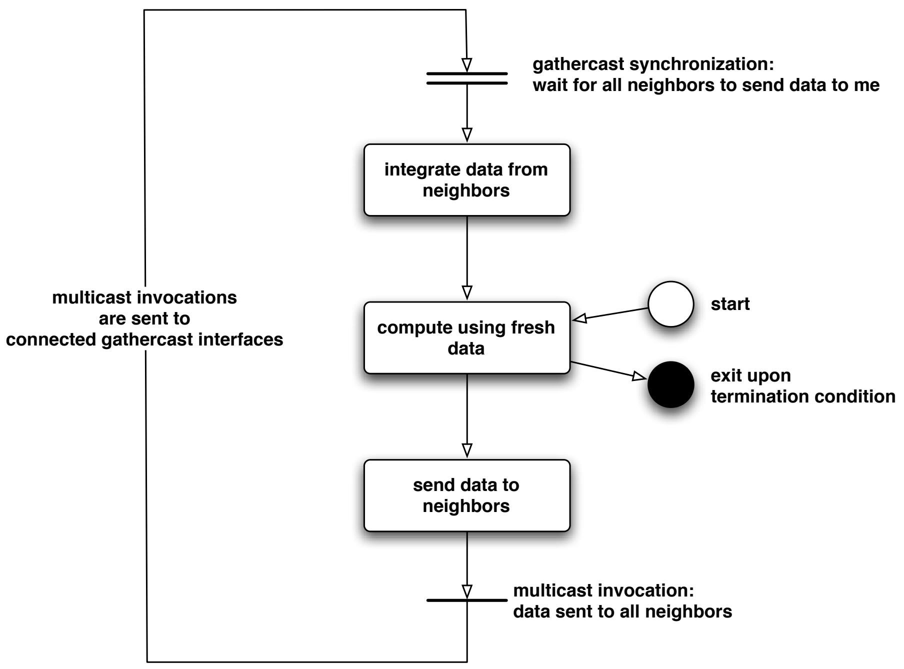

gathercast synchronization:
wait for all neighbors to send data to me
integrate data from neighbors
multicast invocations are sent to connected gathercast interfaces
compute using fresh data
start
exit upon termination condition
send data to neighbors
multicast invocation:
data sent to all neighbors

Figure 4.9: *A typical algorithm for SPMD components*

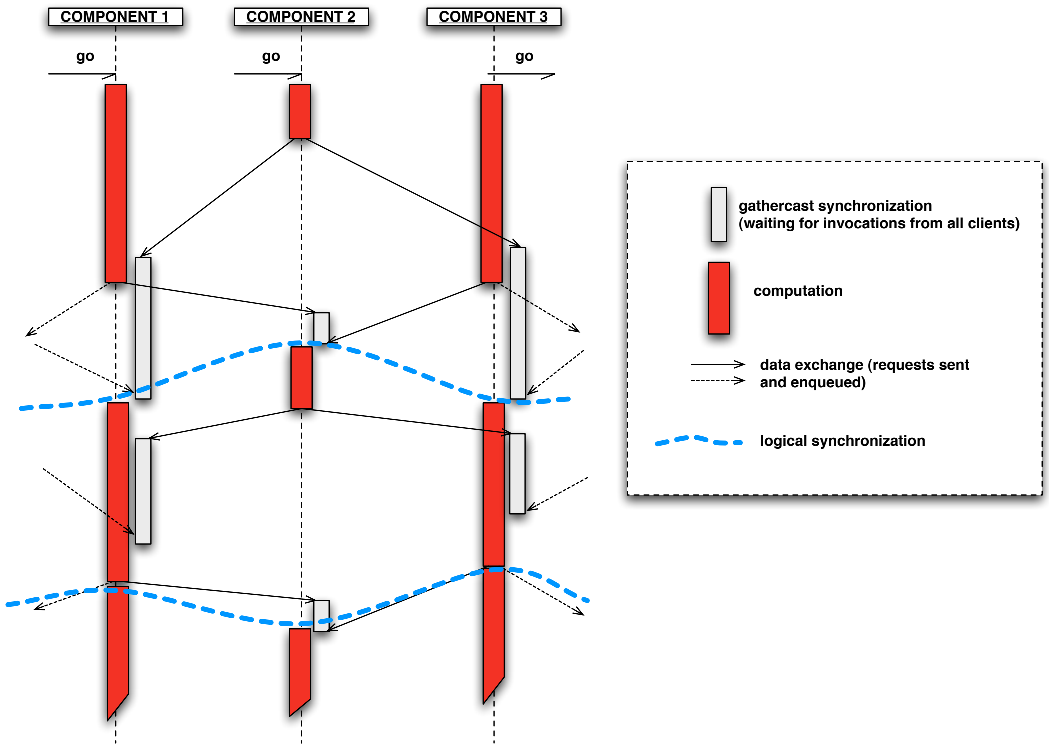

COMPONENT 1
COMPONENT 2
COMPONENT 3
go
go
go
gathercast synchronization (waiting for invocations from all clients)
computation
data exchange (requests sent and enqueued)
logical synchronization

Figure 4.10: *Synchronization between components in a SPMD computation*

have 2, 3 or 4 neighbors. The synchronization between neighbors requires tedious coding efforts. One way is to use master-driven synchronization, in which the computation is driven by a master entity, paying the cost of many extra messages. Another way is to use synchronization barriers, for which a mechanism is available for active objects [BAD 05b], unfortunately barriers complexify the code and require extra synchronization messages which may harm performance. A usually more efficient technique is to use data synchronization, through data buffering techniques, such as odd-even schemes, although the code is generally more complex hence error-prone.

In the approach we propose, the synchronization between components is automatically and transparently handled, through a buffering mechanism. There is no need to write tedious synchronizations in the code of the SPMD entities, provided the functional code follows a standard algorithm such as the one exposed in figure 4.9.

4.4 Towards automatic MxN redistributions

The MxN problem, as defined in section 2.4, refers to the problem of communicating and exchanging data between parallel programs, from a parallel program that contains M processes, to another parallel program that contains N processes. The combination of gathercast and multicast interfaces provides a simple solution to this problem, as exposed in figure 4.11. This solution is a non-optimal one because communications are indirect and data suffers redundant copies.

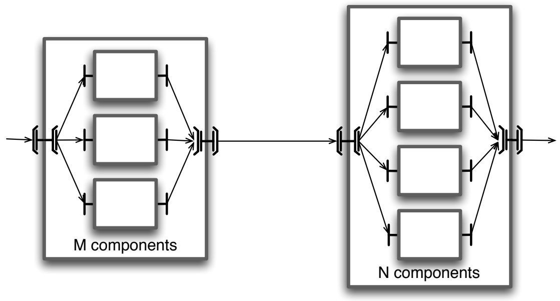


Figure 4.11: *A non-optimized solution to the MxN problem*

The MxN problem is more efficiently addressed in frameworks such as PAWS or GridCCM by *implementation-specific* means. [RIB 04] explicitely defines an abstract model which describes the sequence of operations required by the framework for providing efficient connection between parallel components. Parallel components contain a set of identical components which communicate through an efficient communication layer (either because they are on a same cluster or by using low-level specialized middleware such as MPI). Coupling two parallel components requires first a connection phase, handled by specialized controllers (binding controllers in the case of Fractal). Then a second phase establishes direct communications between internal components of the two parallel components, possibly through a tier communication middleware such as CORBA. A third phase computes the communication schedules for optimizing data redistribution communications using third-party libraries. As a result, the exchange of data is optimized and there is no bottleneck or extra copy and aggregation of parameters.

This abstract model can be projected into the Fractal/ProActive framework by taking advantage of the multicast and gathercast interfaces, and a resulting efficient implementation is modeled in figure 4.12. The implementation would be based on the capabilities of local multicast and gathercast interfaces in order to automatically handle redistribution. The configuration of local multicast and gathercast interfaces could

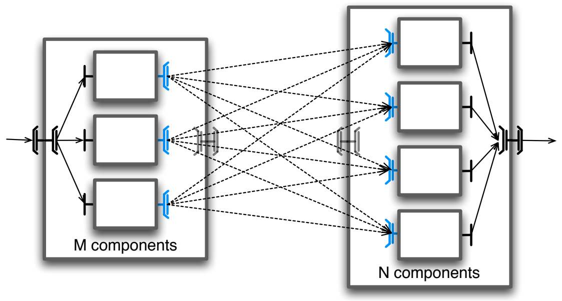


Figure 4.12: *An optimized implementation for the MxN problem based on local multicast and gathercast interfaces*

be *automatically inferred* from the configuration of the collective interfaces involved in the coupling.

We have yet to formally defined gather-multicast interfaces, which offer the features of both gathercast and multicast interfaces, but we note that gather-multicast interfaces would also be an approach to MxN problems with a different set-up (only one side of the participating components are encapsulated in a composite component), as represented in figure 4.13.

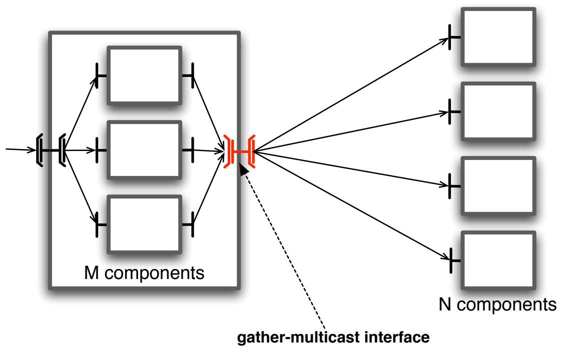


Figure 4.13: *A gather multicast interface for the MxN problem*

4.5 Conclusion

In this chapter, we have proposed a specification for multicast and gathercast collective interfaces. These interfaces may be used for the design of loosely-coupled systems, notably workflows as in Iceni, and we also illustrated how they can leverage component programming by facilitating the design of tightly-coupled applications (SPMD programming style and MxN data redistribution).

As pointed out in various articles [KIE 99, LEE 03], when looking for the best performance, the relationship between the topology of the infrastructure and the collective communications must be taken into account. The combination of virtual node composition and collective interfaces is an attractive and simple way to make sure communications are optimal.

Collective interfaces allow us to refine our previous definition of Grid components.

At the beginning of this thesis, we had envisioned three types of Grid components: primitive, composite, and *parallel* components. We had defined parallel components as composite components redispatching calls to their external server interfaces towards their inner components. The Fractal specification subsequently introduced this notion as an option to the model:

> *A Fractal component may even define a new semantic for the communication between its sub components: instead of specifying that operation invocations follow bindings, [...] it can for example specify that operation invocations are broadcasted to all the sub components, in order to model an asynchronous, reactive "space". Bindings are then useless (another possiblity is to define* parallel components*, where all the sub components have the same type as the enclosing component, and where each operation invocation received on this component is executed in parallel by all its sub components). [BRU 04b].*

However, the semantics of parallel components were not properly specified until the introduction of multicast interfaces, that explicitly define one-to-many typed communications between components. We can now reformulate the definition of a parallel component as follows:

> *A parallel component is a composite component with at least one multicast server interface.*

Invocations received on a multicast server interface of the composite are transformed and redispatched to connected sub-components for a parallel execution. Dispatch can also occur in parallel.

Gathercast interfaces are advantageously used in parallel components to collect and redistribute outputs of the inner components.

We also propose an implementation of collective interfaces: in chapter 5, we describe our implementation of collective interfaces, and in chapter 6, we demonstrate their usability and performance through a series of benchmarks on an SPMD application.

Chapter 5

A Component Framework Based on Active Objects

One of the objectives of this thesis is to propose a framework for developing and deploying component-based applications on Grids. We therefore developed an implementation of the component model described in chapter 3 and 4, based on the ProActive library. Concretely, we implemented the Fractal component model with extensions for parallel computing and specificities related to the active object model. The implementation is named *ProActive/Fractal*.

5.1 Design goals

This framework was designed following five main objectives:

1. Base the implementation on the concept of active objects. The components in this framework are implemented as active objects, and as a consequence benefit from the properties of the active object model.
2. Leverage the ProActive library by proposing a new programming model which may be used to assemble and deploy active objects. Components in the ProActive library therefore also benefit from the underlying features of the library as described in figure 2.14.
3. Provide a component framework suitable for Grid computing, that is, a framework that would answer the requirements as stated in section 1.1. Basing the framework on the ProActive library already allows to fulfill many of these requirements, such as distribution, deployment, scalability and heterogeneity. Bringing the component programming paradigm to the library is a way to tackle the complexity and dynamicity of Grid applications, and to provide higher level concepts for parallel programming.
4. Provide a customizable framework, which may be adapted by the addition of non functional controllers and interceptors for specific needs, and where the activity of the components is also customizable.
5. Implement collective interfaces.

We also propose some optimizations, and as in the Julia implementation, they are achieved to the expense of a trade-off between dynamicity (the possibility to dynamically reconfigure the applications, or parts of the applications) and efficiency (direct or multithreaded invocations).

5.2 Architecture

The ProActive/Fractal framework is an implementation of the Fractal 2 specification. It follows the general model described in the Fractal specification and implements the Fractal Java API. The Fractal specification defines conformance levels for Fractal implementation, categorized from 0 to 3.3. The proposed implementation is conformant up to level 3.3., which means that :

- Components provide a `Component` interface.
- Component interfaces are castable to `Interface`.
- Components with configurable attributes provide the `AttributeController` interface, components with client interfaces provide the `BindingController` interface, components that expose their content provide the `ContentController` interface, and components that expose their life cycle provide the `LifeCycleController` interface.
- A bootstrap component is accessible from a "well-known" name. This bootstrap component provides a `GenericFactory` and a `TypeFactory` interface. Moreover, the `GenericFactory` interface is able to create components with any control interfaces in the set of basic control interfaces (`AttributeController`, `BindingController`, `ContentController`, `LifecycleController`); it can thus create both primitive and composite components. Finally, this interface is also able to create primitive components for which invocations are delegated to the encapsulated implementation code. Note however that some features specific to ProActive are handled in a specific way, notably the lifecycle, detailed later in this chapter.
- The `GenericFactory` interface of the bootstrap component is able to create primitive and composite template components; this is realized by using ADL definitions.

5.2.1 An architecture based on ProActive's Meta-Object Protocol

Our implementation of Fractal relies on ProActive's Meta-Object Protocol architecture.

5.2.1.1 Component instance

A ProActive/Fractal component is an active object. The implementation of a ProActive/Fractal component therefore follows the general architecture represented in figure 2.13. As we stated in the presentation of the ProActive library, the reflective framework may be customized by adding or specializing meta-objects. This allowed us to implement Fractal components using a reflective framework.

A component is instantiated using the standard Fractal API:

```
// get bootstrap component
Component boot = Fractal.getBootstrapComponent();
// get type factory
TypeFactory tf = Fractal.getTypeFactory(boot);
// get generic component factory
GenericFactory gf = Fractal.getGenericFactory(boot);
```

```
// define component type
ComponentType type = tf.createFcType(....);
// define controller description
ControllerDescription controllerDesc = new ControllerDescription(name,
    hierarchicalType);
// define content description
ContentDescription contentDesc = new ContentDescription(implementationClass,
    constructorParameters);

// instantiate component
Component c = gf.newFcInstance(type, controllerDesc, contentDesc);
```

The bootstrap component is retrieved by checking the `fractal.provider` java property (in our implementation,
`org.objectweb.proactive.core.component.Fractive`). The controller part of the component is described in a `ControllerDescription` object. The content part of the component is described in a `ContentDescription` object.

The instance of a component is represented in figure 5.1. The `newFcInstance` method on the component factory returns a Component object. It is a remote reference of type `Component` on the active object which implements the component.

Before describing the architecture of a component in the ProActive library, we first need to clarify the terminology concerning the typing, between objects and components. In the Java language, which follows the object paradigm, the live entities are objects. An object is an instance of a class. The services offered by the class are defined by the methods of this class. In Fractal, which follows the component paradigm, the live entities are instances of components. The services offered by the component are defined by its server interfaces. These server interfaces themselves define methods. Methods are the actual services invoked in implementations of Fractal based on object-oriented languages.

As a consequence, in the object paradigm, an instantiation returns an object of a type compatible with the specified class, whereas in the component paradigm, an instantiation returns a component of type compatible with the specified component type. In the Fractal Java API, a reference on a component is a reference on an object of type `Component`.

Figure 5.1 represents an instance of a ProActive/Fractal primitive component and figure 5.2 represents a composite one.

The design of the implementation of ProActive/Fractal components relies on the general design of active objects represented in figure 2.13. It however exhibits three main specificities. First, a reference on a component from an object A is a reference on a `Component` object, which we can clearly see on the bottom left of the figure. This `Component` object acts as a stub in the standard ProActive architecture, although for performance reasons, a smart proxy pattern is implemented so that common operations, such as getting a reference on a component interface, are performed locally. Using the services of a component implies getting a reference on a given named interface (using the `getFcInterface` method), then invoke methods on this interface. The instance of the `Component` object holds references on local representatives of the functional and non-functional interfaces. These representatives act as stub objects, as they reify invocations and transmit these reified invocations to the proxy. The interfaces representatives are generated dynamically at the creation of the component, or when retrieving a reference on this component through a lookup mechanism. For the sake of clarity, only one of these Interface object is represented on this figure, although all functional and non-functional interfaces are dynamically created locally when creating the reference to the component.

The controller part of the component is implemented as meta-objects as can be seen on the top right of the figures. These meta-objects implement the different controllers, in particular the basic controllers (binding, lifecycle etc...).

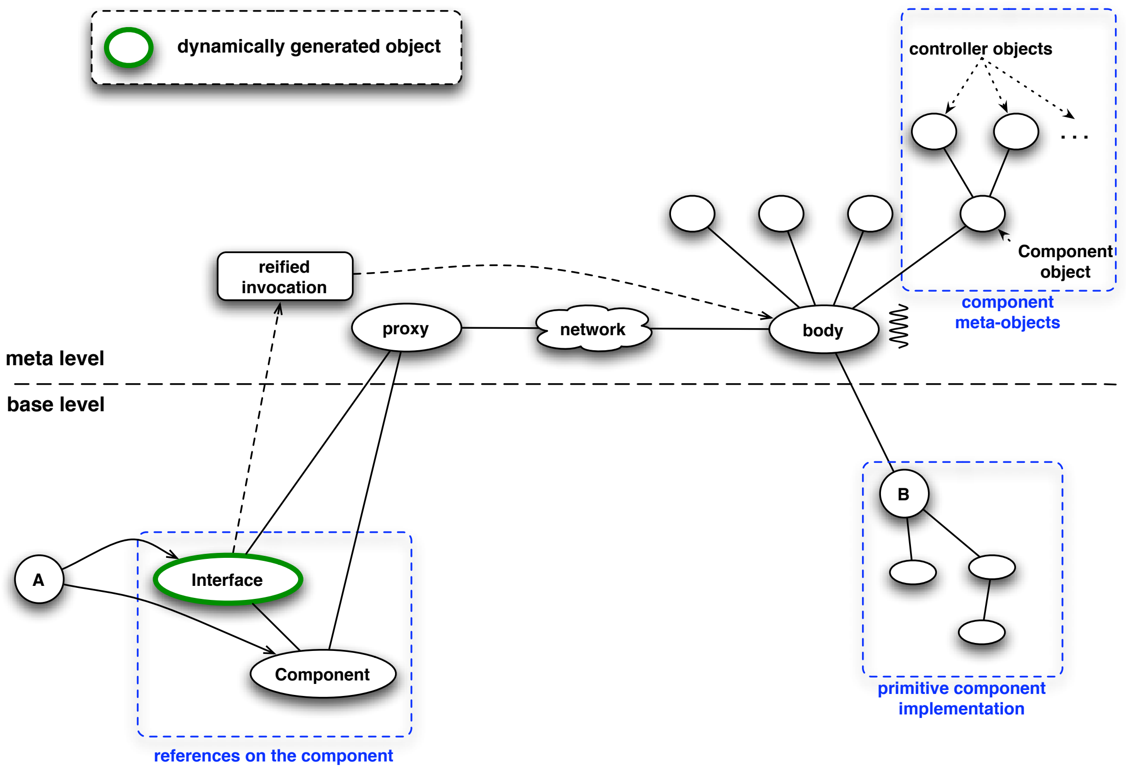


Figure 5.1: *ProActive Meta-Object architecture for primitive components*

Primitive components In a primitive component, the content of the component corresponds to an implementation class, which in ProActive is the root object of the active object, as represented on the bottom right of the figure. Following the Fractal specification, the primitive class may have to implement some callback interfaces such as `BindingController` or `AttributeController`, which are invoked from the meta-level for performing operations which are dependent on the applicative implementation code.

Composite components Figure 5.2 represents an instance of a composite component. A composite component is a structuring component which does not have any business code. Hence the empty composite object as the root of the active object. However, a composite component still offers and requires functional services, and the interfaces objects corresponding to these services are implemented as meta-objects, as represented on the top right of the figure. They may represent internal client interfaces or external client interfaces. A composite component also offers a `ContentController` interface and implementation as a meta-object, for controlling the components it may contain.

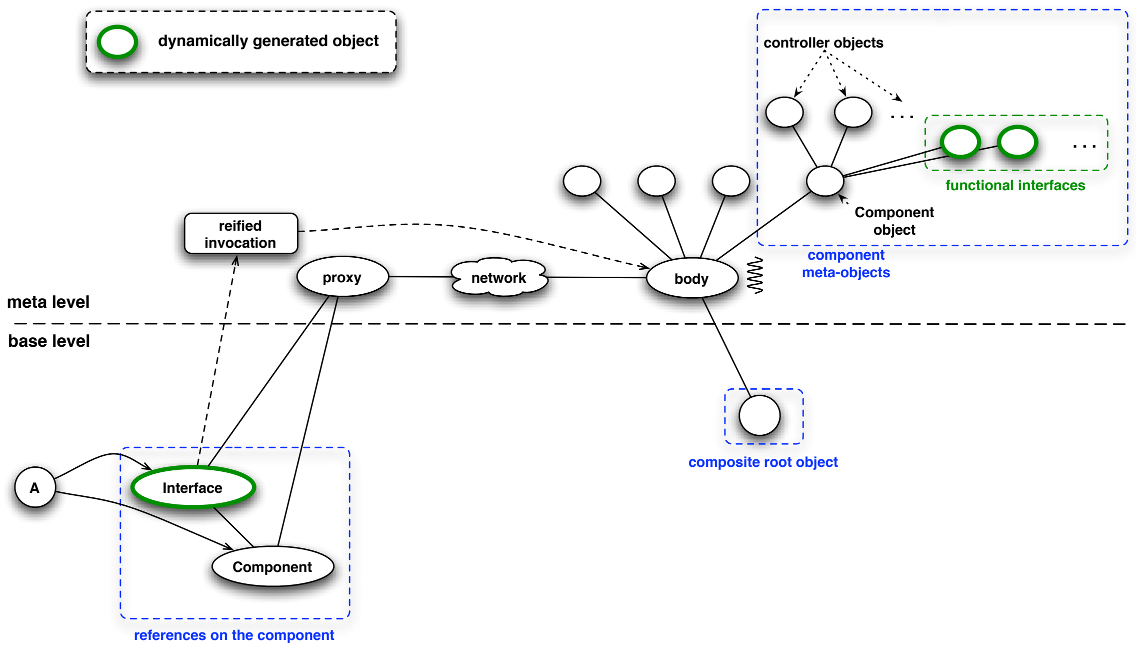


Figure 5.2: *ProActive Meta-Object architecture for composite components*

5.2.1.2 Configuration of controllers

The control part of the component is fully customizable, and the configuration is specified in an XML file, which specifies which control interfaces are offered, and which control classes implement the control interfaces. The default configuration file is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<componentConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="component-config.xsd"
 name="defaultConfiguration">
 <!-- This is the default configuration file for the controllers and interceptors of
      a component in the proactive implementation.-->
   <controllers>
      <controller>
         <interface>org.objectweb.proactive.core.component.controller.
             ProActiveBindingController</interface>
         <implementation>org.objectweb.proactive.core.component.controller.
             ProActiveBindingControllerImpl</implementation>
      </controller>
      <controller>
         <interface>org.objectweb.proactive.core.component.controller.
             ComponentParametersController</interface>
         <implementation>org.objectweb.proactive.core.component.controller.
             ComponentParametersControllerImpl</implementation>
      </controller>
      <controller>
         <interface>org.objectweb.proactive.core.component.controller.
             ProActiveContentController</interface>
```

```
            <implementation>org.objectweb.proactive.core.component.controller.
                ProActiveContentControllerImpl</implementation>
        </controller>
        <controller>
            <interface>org.objectweb.proactive.core.component.controller.
                ProActiveLifeCycleController</interface>
            <implementation>org.objectweb.proactive.core.component.controller.
                ProActiveLifeCycleControllerImpl</implementation>
        </controller>
        <controller>
            <interface>org.objectweb.proactive.core.component.controller.
                ProActiveSuperController</interface>
            <implementation>org.objectweb.proactive.core.component.controller.
                ProActiveSuperControllerImpl</implementation>
        </controller>
        <controller>
            <interface>org.objectweb.fractal.api.control.NameController</interface>
            <implementation>org.objectweb.proactive.core.component.controller.
                ProActiveNameController</implementation>
        </controller>
        <controller>
            <interface>org.objectweb.proactive.core.component.controller.
                MulticastController</interface>
            <implementation>org.objectweb.proactive.core.component.controller.
                MulticastControllerImpl</implementation>
        </controller>
        <controller>
            <interface>org.objectweb.proactive.core.component.controller.
                GathercastController</interface>
            <implementation>org.objectweb.proactive.core.component.controller.
                GathercastControllerImpl</implementation>
        </controller>
        <controller>
            <interface>org.objectweb.proactive.core.component.controller.
                MigrationController</interface>
            <implementation>org.objectweb.proactive.core.component.controller.
                MigrationControllerImpl</implementation>
        </controller>
    </controllers>

</componentConfiguration>
```

We can see the standard required controller interfaces (or extensions of the standard Fractal API control interfaces) for binding, content, name, super. The binding controller is actually only instantiated in case of client interfaces, and the content controller is only instantiated for composite components. The `ComponentParametersController` allows the set-up of specific parameters for this implementation. Some other controllers are related to the features offered by this implementation: migration, management of gathercast and multicast interfaces.

5.2.2 Lifecycle

The lifecycle of components is implemented by customizing the activity of the active objects.

ProActive offers the possibility to customize the activity of an active object; this is actually a fundamental feature of the library, as it allows to fully specify the behavior of active objects. In the context of components, we distinguish the *non-functional* activity from the *functional* activity. The non-functional activity corresponds to the management of component requests when the lifecycle of the component is stopped (i.e. only control requests). The functional activity is encapsulated and starts when the lifecycle is started. This is illustrated in figure 5.3. The default behavior is to serve all control requests in a FIFO order until the component is started, then serve all requests in a FIFO order, until the lifecycle is stopped. The functional activity is encapsulated in the component activity.

It is possible to fully customize this behavior.

First, the functional activity may be customized by implementing the `InitActive`, `RunActive` and `EndActive` interfaces. Two conditions must be respected though, for a smooth integration with the component lifecycle:

1. The control of the request queue must use the `org.objectweb.proactive.Service` class.
2. The functional activity must loop on the body.isActive() condition (this is not compulsory, but it allows to automatically end the functional activity when the lifecycle of the component is stopped. It may also be managed with a custom filter on the request queue).

By default, when the lifecycle is started, the functional activity is initialized, run, then ended upon the !isActive() condition (this method is overriden if the active object is a component, so that this condition is valid when the lifecycle is set to stopped).

Second, the component activity itself may be customized, by implementing the `ComponentInitActive`, `ComponentRunActive` and `ComponentEndActive` interfaces.

5.2.3 Interception mechanism

The Fractal specification states that a component controller can intercept incoming and outgoing operation invocations targeting or originating from the component's sub components. This feature is provided in the ProActive/Fractal implementation, and it allows an interception at the meta-level, of reified invocations, with configurable pre and post method processing. It is an easy way of providing AOP-like features, in order to deal notably with non functional concerns. Interceptors may intercept incoming and outgoing invocations, and they are sequentially composable.

An input interceptor is a controller which must implement the `org.objectweb.proactive.core.component.interception.InputInterceptor` interface, which defines the following methods:

```
public void beforeInputMethodInvocation(MethodCall methodCall);
public void afterInputMethodInvocation(MethodCall methodCall);
```

The `MethodCall` object represents the reified invocation in the ProActive library.

Similarly, an output interceptor must implement the `org.objectweb.proactive.core.component.interception.OutputInterceptor` interface, which defines the following methods:

```
public void beforeOutputMethodInvocation(MethodCall methodCall);
public void afterOutputMethodInvocation(MethodCall methodCall);
```

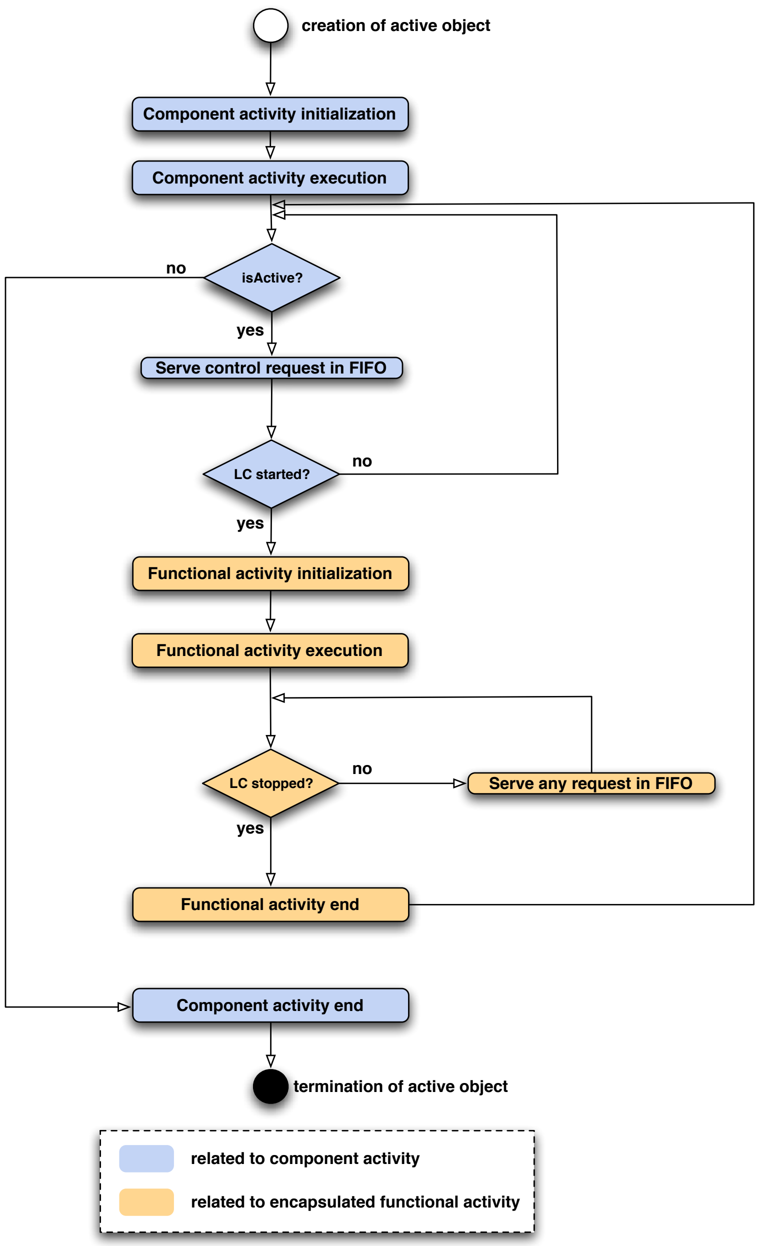

creation of active object
Component activity initialization
Component activity execution
no
isActive?
yes
Serve control request in FIFO
LC started?
no
yes
Functional activity initialization
Functional activity execution
LC stopped?
no
Serve any request in FIFO
yes
Functional activity end
Component activity end
termination of active object
related to component activity
related to encapsulated functional activity

Figure 5.3: *Default encapsulation of functional activity inside component activity*

The input interception mechanism occurs at the service of the request: the reified request is delegated to the input controllers before and after the method is processed.

The output interception mechanism occurs in the interface representative (whose code is dynamically generated) when the invocation is reified: before and after transferring the invocation to the connected component, the reified request is delegated to the output interceptors. The output interception is realized by replacing, during a binding operation, the server interface representative by a server interface representative of the same type, containing the interception code.

Interceptors are configured in the controllers XML configuration file, by simply adding `input-interceptor="true"` or/and `output-interceptor="true"` as attributes of the controller element in the definition of a controller (provided of course the specified interceptor is an input or/and output interceptor). For example a controller that would be an input interceptor and an output interceptor would be defined as follows:

```
<componentConfiguration>
 <controllers>
 ....
   <controller input-interceptor="true" output-interceptor="true">
     <interface>InterceptorControllerInterface</interface>
     <implementation>ControllerImplementation</implementation>
   </controller>
...
```

For input interceptors, the `beforeInputMethodInvocation` method is called sequentially for each controller in the order they are defined in the controllers configuration file. The `afterInputMethodInvocation` method is called sequentially for each controller in the reverse order they are defined in the controllers configuration file.

If in the controller config file, the list of input interceptors declares first, `InputInterceptor1`, and second, `InputInterceptor2`. This means that an invocation on a server interface will follow the path described in figure 5.4. For output interceptors,

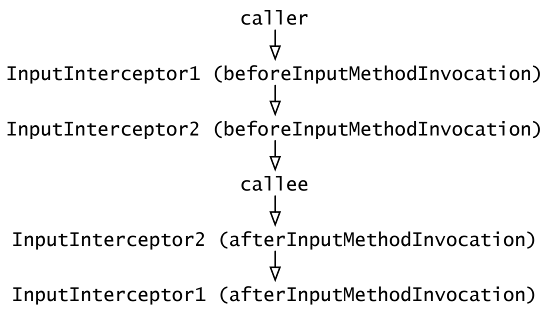

caller
InputInterceptor1 (beforeInputMethodInvocation)
InputInterceptor2 (beforeInputMethodInvocation)
callee
InputInterceptor2 (afterInputMethodInvocation)
InputInterceptor1 (afterInputMethodInvocation)

Figure 5.4: *Execution sequence of an input interception*

the `beforeOutputMethodInvocation` method is called sequentially for each controller in the order they are defined in the controllers configuration file. The `afterOutputMethodInvocation` method is called sequentially for each controller in the reverse order they are defined in the controllers configuration file.

If in the controller config file, the list of output interceptors declares first `OutputInterceptor1` and second `OutputInterceptor2`. This means that an invocation on a client interface will follow the path described in figure 5.5.

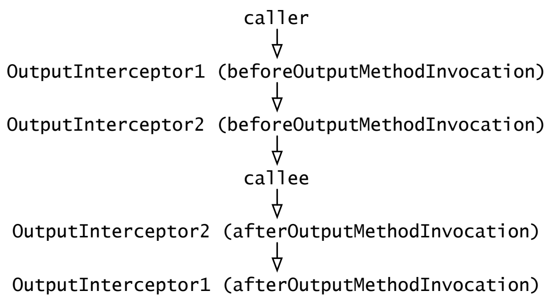

caller
OutputInterceptor1 (beforeOutputMethodInvocation)
OutputInterceptor2 (beforeOutputMethodInvocation)
callee
OutputInterceptor2 (afterOutputMethodInvocation)
OutputInterceptor1 (afterOutputMethodInvocation)

Figure 5.5: *Execution sequence of an output interception*

An interceptor being a controller, it must follow the rules for the creation of a custom controller (in particular, extend `AbstractProActiveController`). Input interceptors and output interceptors must implement respectively the interfaces `InputInterceptor` and `OutputInterceptor`, which declare interception methods (pre/post interception) that have to be implemented.

Here is a simple example of an input interceptor:

```
public class MyInputInterceptor extends AbstractProActiveController
 implements InputInterceptor, MyController {
  public MyInputInterceptor(Component owner) {
    super(owner);
  }

  // some init code
  ...

  // foo is defined in the MyController interface
  public void foo() {
    // foo implementation
  }
  public void afterInputMethodInvocation(MethodCall methodCall) {
    System.out.println("post␣processing␣an␣intercepted␣an␣incoming␣functional␣
         invocation");
    // interception code
  }
  public void beforeInputMethodInvocation(MethodCall methodCall) {
    System.out.println("pre␣processing␣an␣intercepted␣an␣incoming␣functional␣
         invocation");
    // interception code
 }
}
```

The controller is included in the configuration of a given component as follows:

```
<componentConfiguration>
 <controllers>
   ....
   <controller input-interceptor="true">
     <interface>MyController</interface>
     <implementation>MyInputInterceptor</implementation>
   </controller>
   ...
```

5.2.4 Communications

Communications between components in ProActive/Fractal occur through interface references, and rely on the standard ProActive communication mechanism. They may therefore use any underlying protocol supported by ProActive (RMI, RMIssh, http...), and the semantics of invocations are kept, which means that some conditions must be respected for an invocation to be asynchronous. In particular, if the invoked method throws any exception, the invocation is synchronous. This must be considered when using the Fractal API, as most methods of the API declare throwing exceptions. The assembly operations in particular are synchronous operations (although some parallelism may be provided).

Optimization with short cuts Communications between components in a hierarchical model may involve crossing several membranes of enclosing composite components, and therefore paying the cost of several indirections. If the invocations are not intercepted in the membranes, then it is possible to optimize the communication path by short cutting: communicating directly from a caller component to a callee component by avoiding indirections in the membranes.

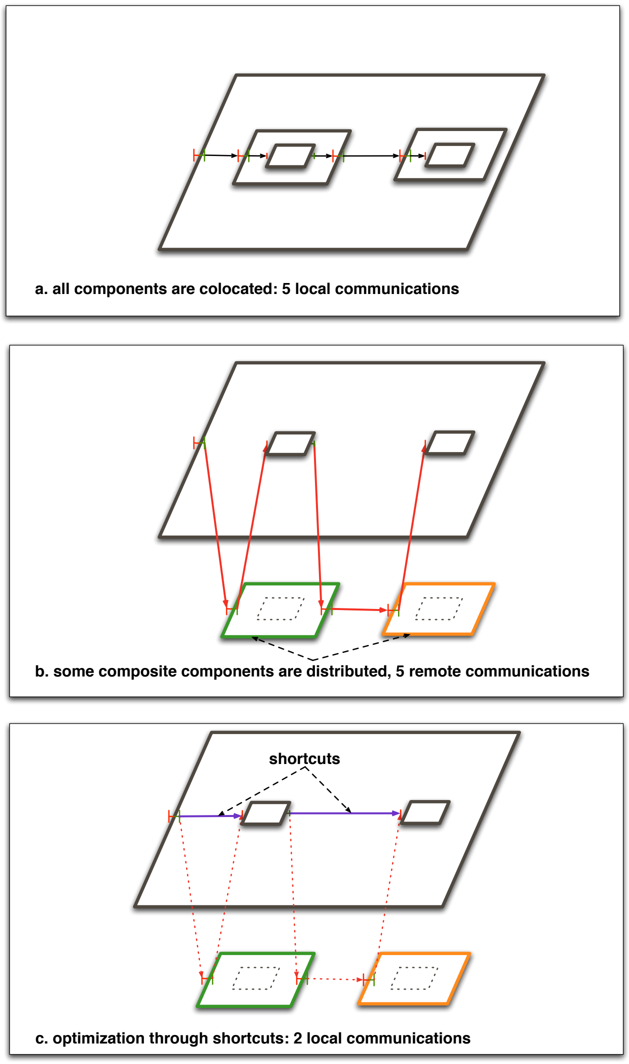

a. all components are colocated: 5 local communications
b. some composite components are distributed, 5 remote communications
shortcuts
c. optimization through shortcuts: 2 local communications

Figure 5.6: *Using short cuts for minimizing remote communications*

In the Julia implementation, a short cut mechanism is provided for components in the same JVM, and the implementation of this mechanism relies on code generation techniques.

We provide a short cut mechanism for distributed components, and the implementation of this mechanism relies on a *tensioning* technique: the first invocation determines the short cut path, then the following invocations will use this short cut path. This mechanism requires composite components to be passive components: objects that augment a root object with a MOP, but do not have any request queue. As a consequence, the rendez-vous of the communication between a client and a server interface, which guarantees causally ordered communications, does not end until the effective server interface has been reached (and the calling thread has returned).

For example, in figure 5.6, a simple component system, which consists of a composite containing two wrapped primitive components, is represented with different distributions of the components. In a, all components are located in the same JVM, therefore all communications are local communications. If the wrapping composites are distributed on different remote jvms, all communications are remote because they have to cross composite enclosing components. The short cut optimization is a simple bypassing of the wrapper components, which results in 2 local communications for the sole functional interface.

Short cuts are available when composite components are synchronous components (this does not break the ProActive model, as composite components are structural components). Components can be specified as synchronous in the `ControllerDescription` object that is passed to the component factory:

```
ControllerDescription controllerDescription =
      new ControllerDescription("name", Constants.COMPOSITE, Constants.SYNCHRONOUS);
```

When the system property `proactive.components.use_shortcuts` is set to true, the component system automatically establishes short cuts between components whenever possible.

Of course, a trade-off is required between performance and flexibility, similarly to the Julia implementation: optimized communications in ProActive/Fractal come at the expense of dynamic reconfigurability.

5.3 Implementation of collective interfaces

In order to provide facilities for parallel programming, ProActive/Fractal proposes an implementation of the proposal of collective interfaces described in chapter 4. To sum it up, the idea is to introduce multicast and gathercast interfaces: multicast interfaces are used for parallel invocations and data redistribution, and gathercast interfaces are used for synchronization and data gathering. The configuration of the collective interfaces policies employs annotations in Java interfaces.

5.3.1 Adaptation of the signatures of methods

As explained in chapter 4, the signatures of methods of client and server interfaces are different when using collective interfaces. For *list* parameters and return types, the possibilities in our implementation are summarized in table 5.1. Broadcast mode is not yet supported for the redistribution of results in gathercast interfaces.

The framework provides transparent adaptation of method invocations and distribution of parameters, through proxies and controllers. Compatibilities of client and server interfaces are checked at runtime, though if using assembly tools this could be checked at design-time.

5.3.2 Multicast interfaces

Our implementation of multicast interfaces does not currently provide dynamic dispatch. The other features described in chapter 4 are available, and in particular, distri-

	client interface	*server interface*
multicast	**List<A> foo (List<B>)**	**A foo (List<B>) (broadcast mode)** **A foo (B) (scatter mode)**
gathercast	**A bar (B) (scatter mode)** ~~**List<A> bar (B) (broadcast mode)**~~	**List<A> bar (List<B>)**

Table 5.1: *Adaptation of method signatures, with list parameters or return types, between client and server interfaces for collective interfaces in the proposed implementation*

bution policies are customizable.

5.3.2.1 Principles

The implementation of multicast interfaces relies on two principles: first, reuse the existing mechanism for typed group communications, and second use a delegation mechanism for adapting the signatures of the interfaces. Therefore, two group proxies are used for a multicast invocation: the first proxy corresponds to the signature of the client interface, and the second to the signature of the server interfaces. Bindings are transparently handled so that the client component receives a reference on a group proxy of the type of the client interface.

This mechanism is illustrated in figure 5.7, which corresponds to the design represented in figure 5.8.

When an invocation is performed, a reified invocation is first created (here on method `void bar(List<A>)`), given to the first group proxy, which delegates it to a second proxy of the type of the server interfaces (for invocations on method `void bar(A)`). Parameters are then automatically distributed according to the distribution policy specified as an annotation, and the second proxy transfers the new reified invocations to connected server interfaces in a parallel manner (using the standard multithreading mechanism of ProActive typed groups). This delegation and adaptation process between group proxies is implemented by extending the standard group proxy, the `ProxyForGroup` class, into the `ProxyForComponentInterfaceGroup`.

5.3.2.2 Configuration

The distribution of parameters in our framework is specified in the definition of the multicast interface, using annotations. Elements of a multicast interface which can be annotated are: interface, methods and parameters. The different distribution modes are explained later. The examples in this section all specify broadcast as the distribution mode.

Interface annotations A distribution mode declared at the level of the interface defines the distribution mode for all parameters of all methods of this interface, but may be overriden by a distribution mode declared at the level of a method or of a parameter. The annotation for declaring distribution policies at level of an interface is `@org.objectweb.`

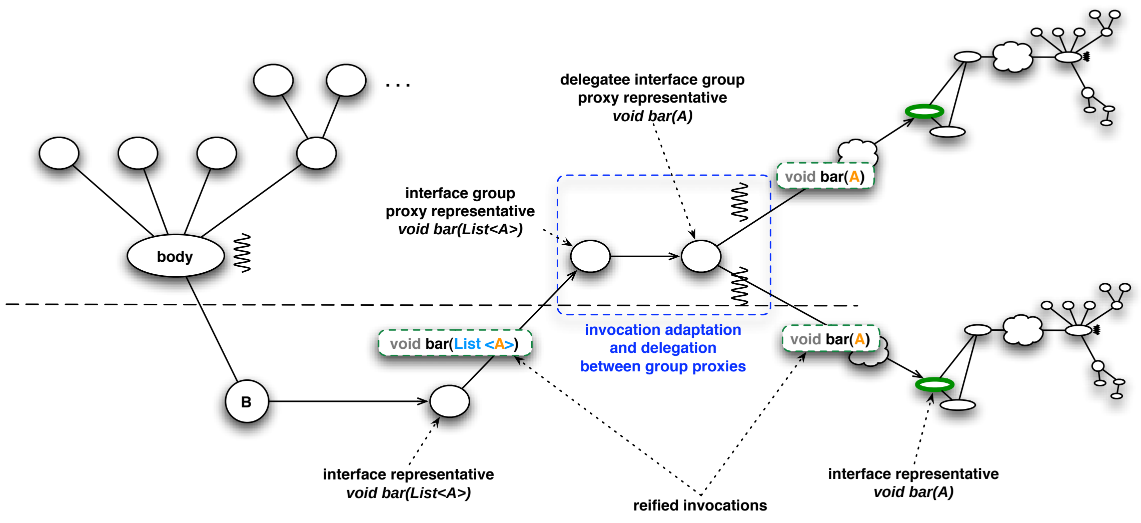


Figure 5.7: *Adaptation and delegation mechanism for multicast invocations*

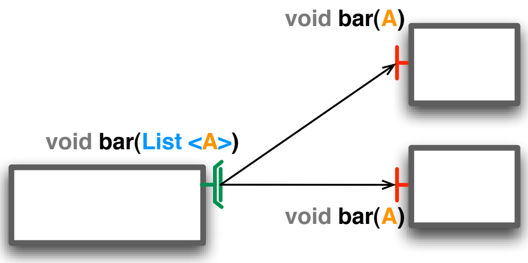


Figure 5.8: *An example of multicast interfaces: the signature of an invoked method is exposed, and in this case exhibits a scattering behavior for the parameters*

proactive.core.component.type.annotations.multicast.ClassDispatchMetadata and is used as follows:

```
@ClassDispatchMetadata(mode=@ParamDispatchMetadata(mode=ParamDispatchMode.BROADCAST)
    )
interface MyMulticastItf {
public void foo(List<T> parameters);
}
```

Method annotations A distribution mode declared at the level of a method defines the distribution mode for all parameters of this method, but may be overriden at the level of each individual parameter. The annotation for declaring distribution policies at level of a method is @org.objectweb.proactive.core.component
.type.annotations.multicast.MethodDispatchMetadata and is used as follows:

```
@MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode =ParamDispatchMode.
    BROADCAST))
public void foo(List<T> parameters);
```

Parameter annotations The annotation for declaring distribution policies at level of a parameter is `@org.objectweb.proactive.core.component.type.annotations .multicast.ParamDispatchMetadata` and is used as follows:

```
public void foo(@ParamDispatchMetadata(mode=ParamDispatchMode.BROADCAST) List<T>
    parameters);
```

For each method invoked and returning a result of type `T`, a multicast invocation returns an aggregation of the results: a `List<T>`. There is a type conversion, from return type `T` in a method of the server interface, to return type `List<T>` in the corresponding method of the multicast interface. The framework transparently handles the type conversion between return types, which is just an aggregation of elements of type `T` into a structure of type `List<T>`.

Available distribution policies 3 modes of distribution of parameters are provided by default, and define distribution policies for lists of parameters:

- `BROADCAST` copies a list of parameters and sends a copy to each connected server interface. The distribution of parameters corresponds to the following transformation for all invocations (one for each connected server interface):
  $foo(a_1, ...a_m, L_1, ...L_l) \longrightarrow foo(a_1, ...a_m, L_1, ...L_l)$
- `ONE-TO-ONE` sends the i^{th} parameter to the connected server interface of index i. This implies that the number of elements in the annotated list must be equal to the number of connected server interfaces. The distribution of parameters corresponds to the following transformation for the i^{th} invocation, sent to the i^{th} connected server interface:
  $foo(a_1, ...a_m, L_1, ...L_l) \longrightarrow foo(a_1, ...a_m, L_{1_i}, ...L_{l_i})$
- `ROUND-ROBIN` distributes each element of the list parameter in a round-robin fashion to the connected server interfaces. The distribution of parameters corresponds to the following transformation:
  $0 \leq i \leq n : foo(a_1, ...a_m, L_1, ...L_l) \longrightarrow foo(a_1, ...a_m, L_{1_{card(L_1) mod(i)}}, ...L_{l_{card(L_l) mod(i)}})$
  with n the number of connected server interfaces.

It is also possible to define custom distributions by specifying the distribution algorithm in a class which implements the `org.objectweb.proactive.core.component.type. annotations.multicast.ParamDispatch` interface, thereby defining the distribution algorithm which will be used during the dispatch phase. There are only three methods to implement:

```
public List<Object> dispatch(Object inputParameter, int nbOutputReceivers) throws
    ParameterDispatchException;

public int expectedDispatchSize (Object inputParameter, int nbOutputReceivers)
    throws ParameterDispatchException;

public boolean match(Type clientSideInputParameter, Type serverSideInputParameter)
    throws ParameterDispatchException;
```

Then the custom dispatch mode is used as follows:

```
@ParamDispatchMetadata(mode =ParamDispatchMode.CUSTOM,
          customMode=CustomParametersDispatch.class))
```

5.3.3 Gathercast interfaces

The implementation of gathercast interfaces in our framework is restricted to the management of a basic synchronization. Synchronization policy is not configurable, except for a timeout which can be specified if the method returns a result. Data redistribution policies for results are not configurable and the redistribution of results occurs in a one-to-one manner to the client interfaces.

5.3.3.1 Principles

Bindings to gathercast interfaces are bi-directional (see section 4.2.2.3), which means that the server gathercast interface holds a reference to its clients. This is used for synchronization: once an invocation on a given method `foo1` comes from a client interface, the gathercast interface will create the corresponding request to be processed by the server component until all clients have sent an invocation on this method foo1. Until this condition is reached, the requests are queued in a special queue.

At the moment the reified invocation on method `foo1` from the last connected client is served, the synchronization condition is reached, and a new reified invocation is created by gathering all parameters from all client invocations. The new reified invocation is then served by the server component.

The data structures representing the queues of requests (reified invocations) is illustrated in figure 5.9. In the figure, we can see the enqueued requests for one gathercast interface (`gathercastItf1`) and two different methods. Suppose we have three clients, and *Ri* is an incoming request from client *i*. In the case of `foo1`, when the request on this method coming from client 3 will be served, then a new request will be created and served by the component, and the queue will be emptied of the corresponding requests. We can also observe that client 1 invoked `foo1` twice, but the mechanism waits for the first queue to be full until processing any other queue, even though they are full. This is a way to guarantee causal dependency.

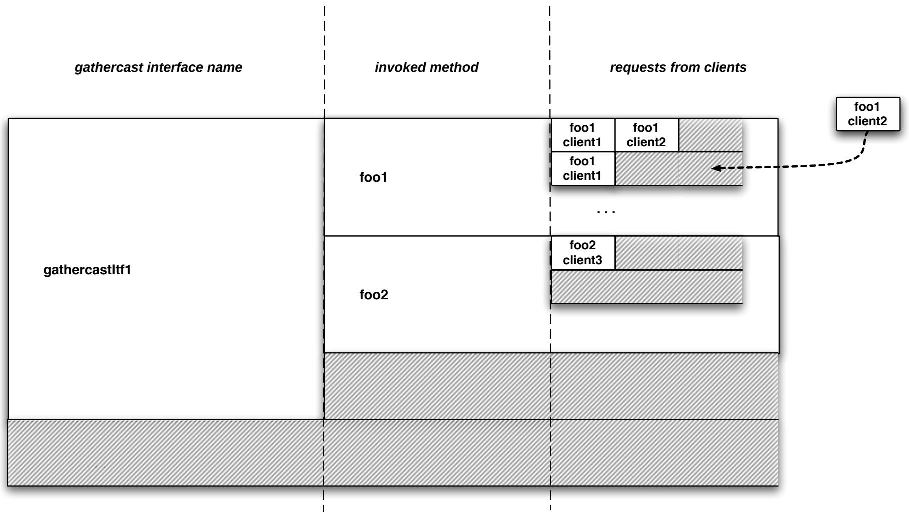


Figure 5.9: *Data structure for the buffering of requests in gathercast interfaces*

5.3.3.2 Asynchronism and management of futures

A fundamental feature of the library is the asynchronism of method invocations: we want to preserve it in the context of gathercast interfaces, not only between client and server gathercast interface, but also for the transformed invocation in the gathercast interface.

When the invoked method returns void, there is no problem as this is considered as a one way invocation in ProActive, no future result is expected. If the invoked method returns a result however, the method returns a future, although the invocation has not been processed yet (an invocation on a gathercast interface will not proceed until all client interfaces invoked the same method). We faced a complex problem: how to return and update futures of client invocations on gathercast interfaces? We considered two strategies. The fist one was to customize the request queue so that a local data structure (similar to the one described in figure 5.9) would handle the incoming requests for gathercast interfaces. A second option was to use a dedicated tier active object for handling futures.

As we did not want to intervene in the core of the ProActive library by modifying the request queue, we selected and implemented the second option, which not only works but also provides an example of the management of futures and automatic continuations with active objects.

The mechanism is illustrated in figure 5.10. One futures handler active object is created for each gathercast request to be processed. It has a special activity, which only serves `distribute` requests once it has received the `setFutureResult` request.

When a request from a client is served by the gathercast interface, it is enqueued in the queue data structure, and the result which is return is the result of the invocation of the `distribute` method (with an index) on the futures handler object. This result is therefore a future itself.

When all clients have invoked the same method on the gathercast interface, a new request is built and served, which leads to an invocation which is performed either on the base object if the component is primitive, or on another connected interface if the component is composite. The result of this invocation is sent to the futures handler object, by invoking the `setFutureResult` method. The futures handler will then block until the result value is available. Then the distribute methods are served and the values of the futures received by the clients are updated.

Although this mechanism fulfills its role using the standard mechanism of the library, we observed that it does not scale very well: one active object for managing futures is created for each gathercast request, and even though we implemented a pool of active objects, there are too many active objects created when stressing the gathercast interface. Therefore, the first approach envisaged above should be preferred in the future (we unfortunately did not have time to implement it, as it is quite complex and deals with sensitive parts of the library).

5.3.3.3 Timeout

It is possible to specify a timeout, which corresponds to the maximum amount of time between the moment the first invocation of a client interface is processed by the gathercast interface, and the moment the invocation of the last client interface is processed. Indeed, the gathercast interface will not forward a transformed invocation until all invocations of all client interfaces are processed by this gathercast interface. Timeouts

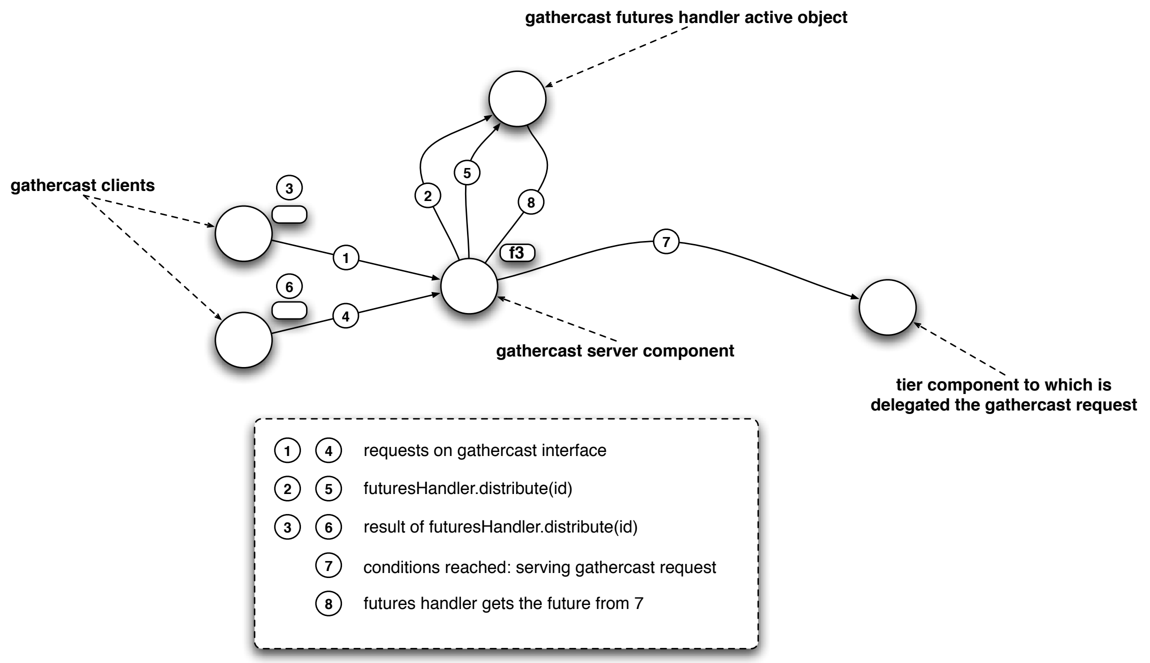

gathercast futures handler active object
gathercast clients
gathercast server component
tier component to which is delegated the gathercast request
f3
1 4 requests on gathercast interface
2 5 futuresHandler.distribute(id)
3 6 result of futuresHandler.distribute(id)
7 conditions reached: serving gathercast request
8 futures handler gets the future from 7

Figure 5.10: *Management of futures for gathercast invocations*

for gathercast invocations are specified by an annotation on the method subject to the timeout, the value of the timeout is specified in milliseconds:

```
@org.objectweb.proactive.core.component.type.annotations.gathercast.MethodSynchro(
    timeout=20)
```

If a timeout is reached before a gathercast interface could gather and process all incoming requests, a `org.objectweb.proactive.core.component.exceptions.GathercastTimeoutException` is returned to each client participating in the invocation. This exception is a *runtime exception*. Timeouts are *only applicable to methods which return a non-void value*: there is no simple way otherwise to inform the client that the timeout has been reached: the client would need to provide a callback interface, which does not fit well with a simple invocation-based programming model.

5.4 Deployment

The deployment process is based on both the Fractal ADL capabilities and the ProActive deployment framework. A component system is usually described using an ADL, and the location of the components are specified in the ADL using the virtual node abstraction. Virtual nodes are then mapped to the physical infrastructure by using the standard ProActive deployment mechanism as described in 2.6.2.5.

5.5 Conclusion

During this thesis, we developed a implementation of the Fractal model based on the ProActive library; this implementation supports extensions for parallel computing, and is highly configurable. It offers some optimizations and facilities for parallel computing which address specificities of Grid computing.

This framework is the result of several years of developments and experiments. It is quite complex and uses many engineering techniques such as bytecode generation, synchronization, multithreading, code annotations and reflection. The core of the ProActive/Fractal framework contains around 17000 lines of code, located in 164 classes in the package `org.objectweb.proactive.core.component` and sub-packages. There is minimal intrusion in the core of the ProActive library. The framework is thoroughly tested through 22 regression tests of fine and medium grain size.

The framework is extensible, and it is released in open-source as part of the ProActive library: it may be freely downloaded and used. It is distributed with user documentation, developer documentation as javadoc, and a number of examples, including a version of the Hello World example from the Julia implementation incremented with deployment features.

The next chapter describes some of the applications developed with this framework: it discusses methodology for component-based programming and presents some applications that were used to evaluate usability, scalability and performance of the framework.

Chapter 6

Methodology and Benchmarks

This chapter relates our experiments with the ProActive/Fractal framework; it is divided in three sections. First, we evaluate the feasibility of transforming existing applications with an object-oriented design towards a component-oriented design. Second, we outline the benefits of such a refactoring, and considering a relatively complex application, we describe a methodology for refactoring existing applications, and we evaluate the performance of the resulting component-based version with respect to the object-oriented one. We finish in the third section, by evaluating the usage of collective interfaces for SPMD programming.

6.1 C3D

The C3D benchmark is a Java benchmark application that measures the performance of a 3D raytracer renderer distributed over several Java virtual machines using Java RMI. This benchmark gives an indication of the performance of the serialization process and of Java RMI itself.

Our C3D application is a user-driven version of the benchmark, and is both a collaborative application and a distributed raytracer: users interact through messaging and voting facilities in order to choose a scene that is rendered using a set of distributed rendering engines working in parallel.

We refactored the C3D application as a proof-of-concept for using a component-based approach.

6.1.1 Methodology

The core idea was to minimize the modifications of the existing code. For this purpose, we used inheritance and created the required explicit interfaces of a component-based design when necessary. The approach taken is similar to the approach presented in the Fractal tutorial [FRAb]. This experiment allowed us to experiment our framework based on active objects with an existing interactive application.

6.1.2 Design

C3D is designed around three core entities: the first one is the *users*, which interact and vote for updating a 3D scene, The second one is the *dispatcher*, which acts as the election referee and delegates the update orders to available rendering engines. The third one is the rendering engines, which perform a parallel computation of the scene to render, and send back the results to the dispatcher, which itself updates the users. Each one of these entities is implemented as an active object, which we want to view as a component.

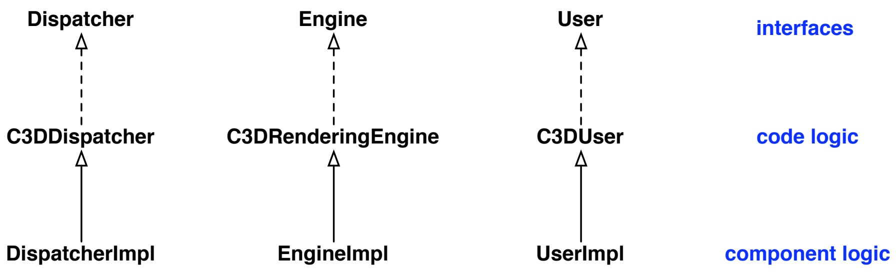


Figure 6.1: *C3D component classes and interfaces*

We chose a wrapping approach for refactoring the application, in order to keep the original code. Wrapping consists in encapsulating the original classes. Instead of linking the classes together by setting fields through the initial methods, this is done in binding methods, using the inversion of control pattern. For this wrapping task, we had to localize the assignation of references to the `C3DRenderingEngine`, `C3DUser` and `C3DDispatcher` classes, and *expose these assignations as component bindings*. We also need to identify and expose functional interfaces, hence the `Dispatcher`, `Engine` and `User` interfaces which must be implemented by the wrapping classes. The resulting class design is represented in figure 6.1, and the components are described in ADL files. User components can be instantiated and dynamically bound to the application through a lookup mechanism.

6.1.3 Observations

Refactoring a basic application is a good proof of concept that provided us with preliminary feedback on the component framework implementation and usability. The immediate benefits that we identified are a better modularity thanks to the absence of explicit dependencies between entities, as well as an enforced separation of interface and implementation, a requirement for easily updating or replacing code.

We also identified the C3D application as a promising candidate for experimenting gathercast selection strategies in the context of collaborative applications. The current C3D application performs a selection of the inputs of users based on a voting mechanism, which is entangled with the functional code of the dispatcher. Further refactoring will connect the users and dispatcher components with a gathercast interface. Using selection strategies with gathercast interfaces will separate the voting logic and allow a customization of the selection parameters.

6.2 Jem3D

In this experiment, we refactored an existing application towards a component-oriented design and evaluated qualitatively and quantitatively (through benchmarks) the resulting implementation.

6.2.1 The Jem3D application

Jem3D is a numerical solver for the 3D Maxwell's equations modelling the time domain propagation of electromagnetic waves. It relies on a finite volume approximation method designed to deal with unstructured discretization of the computation domain. A domain is divided into tetrahedra where local calculation of the electromagnetic fields is performed. More precisely, the balance flux is evaluated as the combination of the elementary fluxes computed through all four facets of a tetrahedron. After each computation step, the local values calculated in a tetrahedron are passed to its neighbours and a new local calculation starts.

The intensity of the computation can be changed by modifying the number of tetrahedra in the domain. This is done by setting the mesh size, that is, the triplet (m1;m2;m3) that determines the number of points on the x, y, and z axis used for the building of the tetrahedral mesh. The higher the mesh size, the higher the number of tetrahedra, and the more computation intensive the application.

Parallelization in Jem3D relies on dividing the computational domain into a number of subdomains; this division is specified by another triplet that determines the number of subdomains on the x, y, and z axis. Since some facets are located on the boundary between subdomains, neighbouring subdomains must communicate to compute the values of those border facets. More details on Jem3D can be found in [HUE 04].

Figure 6.2 shows the runtime structure of the original Jem3D (a 2*2*1 division into subdomains is assumed); the main elements of the architecture are outlined next.

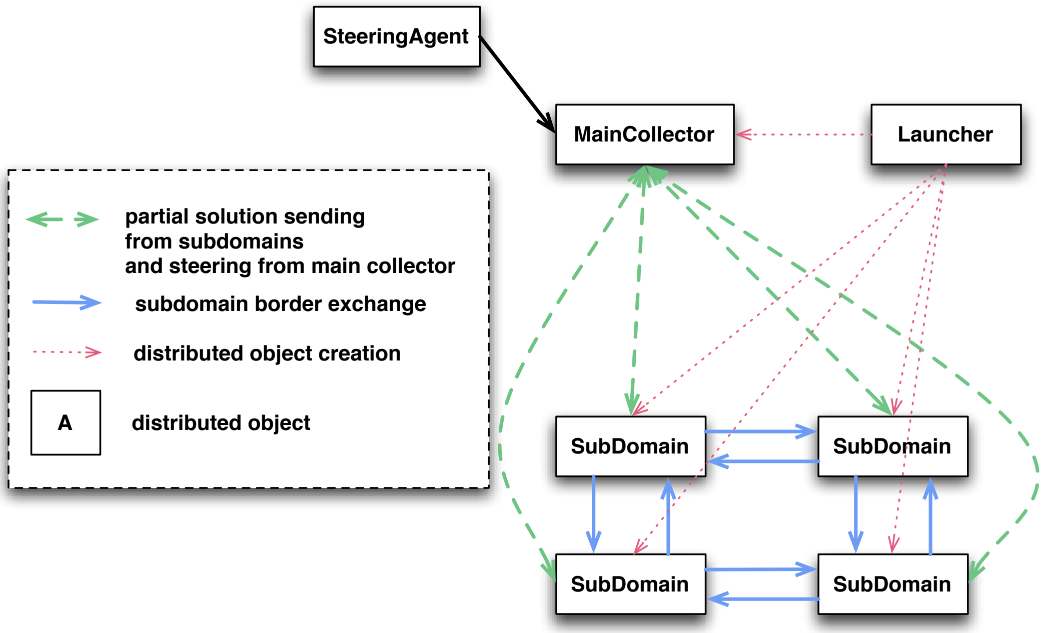

SteeringAgent
MainCollector
Launcher
partial solution sending from subdomains and steering from main collector
subdomain border exchange
distributed object creation
A
distributed object
SubDomain
SubDomain
SubDomain
SubDomain

Figure 6.2: *Original Jem3D architecture following an object-oriented design*

Subdomains correspond to partitions of the 3D computational domain; they perform electromagnetic computations and communicate with their closest neighbours in the 3D grid. Moreover, they send partial solutions with a predefined frequency to the main collector. The *main collector* is responsible for monitoring and steering the computation by interacting with the subdomains. The monitoring and steering functionality is used by one or more steering agents, which are dynamically registered with the main collector.

The application includes a command-line agent and a graphical agent with visualisation capabilities. *Steering agents* communicate with each other to ensure that only a single agent at a time has the right to control the computation. Finally, the *launcher* is responsible for obtaining the input data, creating the main collector and the subdomains, setting up the necessary connections between them, initialising them with the necessary information, and starting the computation. Communication between the entities relies on the asynchronous remote invocation and group communication mechanisms provided by ProActive.

6.2.2 Problem statement

The original Jem3D application suffers from limited modifiability and limited reusability of its parts, which can be largely attributed to two factors.

First, the application lacks reliable architectural documentation, which is essential for understanding and enhancing complex software systems. Jem3D has been subjected to successive changes by multiple people without corresponding updates to the architectural information, which has thus become obsolete. Since the original object-based design enforced neither separation of interfaces from implementation, nor separation of concerns, the software architecture suffered from erosion.

Second, the application parts are tightly coupled together. Indeed, as in most object-oriented applications, the code includes hard-wired dependencies to classes, which limits their reusability, increases the impact of changes, and inhibits run-time variability. For example, changing the subdomain implementation requires updating the source code of both the main collector and the launcher and rebuilding the whole application. As another example, although the Jem3D parallelisation follows a typical geometric decomposition pattern, no part of the application can be reused in other contexts where this pattern is applicable.

To address such modifiability and reusability limitations, Jem3D was re-engineered into a component-based system.

6.2.3 Methodology

This section presents our approach for addressing the modifiability and reusability limitations of Jem3D. The approach consists of a general componentization process and the use of the Fractal/ProActive component technology, which are discussed in the following two sections.

Componentization process The purpose of the componentization process is to transform an object-based system into a component-based system. The process assumes that the target component platform allows connecting components via provided and required interfaces, and that it minimally supports the same communication styles as the object platform (e.g., remote method invocation and events). Figure 6.3 shows the main activities and artefacts defined by the componentization process. Note that the activities do not necessarily proceed sequentially. For example, the activity "Implement interface-based version" may be performed when an initial component architecture is created and revisited when an updated architecture is available. The activities of the process are explained next.

A. Recovery of original architecture The goal of this activity is to produce an architectural description of the original system, serving as the basis for understanding

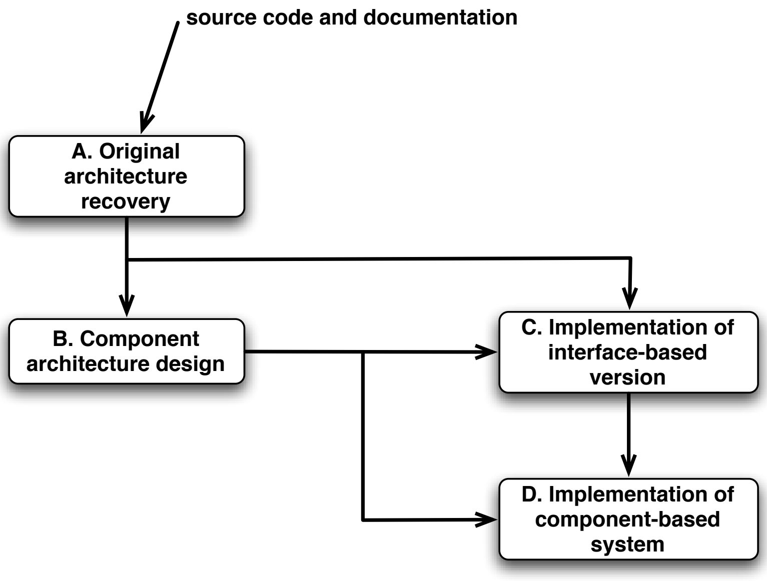

source code and documentation
A. Original architecture recovery
B. Component architecture design
C. Implementation of interface-based version
D. Implementation of component-based system

Figure 6.3: *Componentization process*

and transforming the system. The activity uses as input the source code, documentation, build files, and any other software artefacts. It consists in analysing the source system, extracting architecturally significant information, and documenting different views of the architecture. As a minimal requirement, the documentation must include a run-time view describing executing entities (e.g., objects or distributed objects), communication paths, and interactions over those paths (e.g., sequences of remote method invocations).

B. Component architecture design The goal of this activity is to design the target component architecture using as input the original architecture. The component architecture specifies a set of components, their relationships, and the interactions among them and builds on the target component model. The activity can be divided into four steps:

- *Define initial architecture*. The executing entities of the original architecture are used as candidate components to form an initial component architecture.
- *Refine component selection*. Candidate components are decomposed into smaller components or integrated into larger components, and their relationships and interactions are updated accordingly. These changes are driven by modifiability and performance concerns. Decomposition is typically used to increase the reusability of components and the flexibility of the architecture, whereas integration is used to reduce performance overheads.
- *Specify component interfaces*. By analyzing and organizing the interactions between each component and its environment, this step identifies provided and required interfaces. Multiple interfaces for each component are defined in order to reduce dependencies.
- *Refine architecture using available component model features*. The component architecture is adapted to exploit all the available features provided by the tar-

get component model, such as hierarchical composition in Fractal, or implicitly-accessed, container services in CCM.

C. Implementation of interface-based version The goal of this activity is to implement and test an interface-based version of the system in which entities are connected via provided/required interfaces. The activity involves restructuring the original code without adding dependencies on the target component model. The motivation for this activity is to validate an important part of the target component architecture at an earlier time, before migrating the code to the component platform. Moreover, this activity simplifies this migration because of the closer correspondence of the produced code to the target architecture. The activity can be divided into the following steps:

- *Align code with component architecture*. This step ensures that the code includes classes which correspond to all intended components, and that these classes implement all interfaces provided by their corresponding components.
- *Add dependency injection mechanism*. Supporting configurable connections requires a uniform mechanism for injecting references to required interfaces into objects. Such mechanisms are provided by most component models, and are manifested as standard methods for accepting and managing interface references. This step ensures that all classes corresponding to intended components support an injection mechanism, thus making their dependencies explicit and externally modifiable.
- *Use injection mechanism*. This step modifies the classes so that they invoke collaborating classes only through injected references. Moreover, the step modifies any "injector" code that supplies a class with references to required objects to use the uniform injection mechanism.

D. Implementation of component-based system The goal of this activity is to implement and test the new component-based system. It uses as inputs the component architecture and the intermediate, interfacebased version. It typically involves minor changes for repackaging classes as component implementations. It may also involve changes for exploiting features of the component model that were unavailable in the original object platform.

Most of the effort was spent on the architecture recovery activity because of the undocumented and degraded structure of the system. The run-time view of the original architecture is described in figure 6.2. During the component architecture design, the launcher component was decomposed into a subdomain factory component and an activator component; the launcher is responsible for creating, initialising, and connecting the subdomains, and the subdomain factory is responsible for obtaining the input data, passing them to the factory, and starting the computation. The reason for the decomposition was to make the factory reusable beyond Jem3D.

A later iteration of the activity grouped the factory and the subdomains into a composite domain component, exploiting the hierarchical composition feature of Fractal/ProActive. Implementing the interface-based version served to increase confidence in the new component architecture, and drastically simplified the final component-based implementation. The component-based implementation involved wrapping classes to form Fractal components and replacing a large part of injector logic with Fractal ADL descriptions (see below).

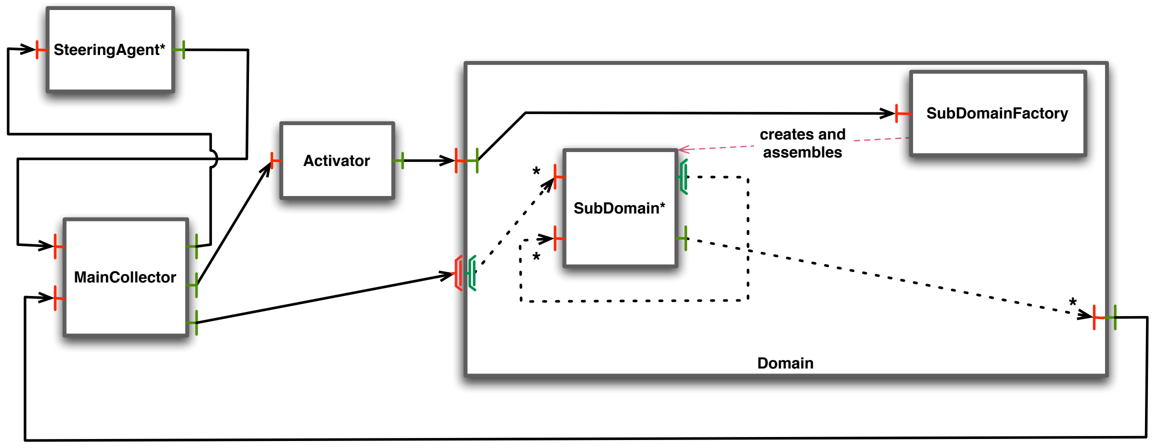


Figure 6.4: *Component-based Jem3D design*

Figure 6.4 shows the static structure of the resulting component-based Jem3D using a Fractal representation, which was modified to include cardinality attributes, as in UML for instance. Indeed, the number of subdomains is variable and it would be clumsy to represent without an abstraction for multiple instances. The runtime configuration consists of multiple subdomains, logically arranged in a 3D mesh, with each subdomain connected to its neighbours via multicast interfaces. The runtime configuration also includes a dynamically-determined number of steering agents. The main collector is connected to all agents via a multicast interface; agents are connected to each other via multicast interfaces.

The initial configuration of Jem3D is described using Fractal ADL, and a version written in pseudo-code is presented here:

```
component ConsoleSteeringAgent definition = SteeringAgentImpl
component MainCollector definition = MainCollectorImpl
component Activator definition = ActivatorImpl
component Domain
      interface ... // interfaces definitions omitted
      component SubDomainFactory definition=FactoryImpl (SubDomainImpl)
      // bindings within composite
      // interfaces names omitted
      binding this --> SubDomainFactory

// bindings among top-level components
// interface names omitted
binding ConsoleSteeringAgent --> MainCollector
binding MainCollector --> ConsoleSteeringAgent
binding Activator --> MainCollector
binding Activator --> Domain
binding MainCollector --> Domain
binding Domain --> MainCollector
```

The ADL is not used to express the configuration of subdomains, which depends on the dynamically-determined subdomain partitioning. Since allowable configurations follow a fixed, canonical structure in the form of a 3D mesh, a parameterized description would be useful for automatically generating subdomain configurations. Currently, the basic parameterization mechanism included in Fractal ADL is simply used to configure the factory with the required subdomain implementation.

6.2.4 Evaluation of the new design

We now examine whether the new, component-based Jem3D addresses the modifiability and reusability limitations of the original system. Owning to the componentization process, the new system has a reliable architectural documentation, which facilitates the understanding and the evolution the system. Moreover, an important part of the architecture - i.e., the initial component configuration - is now captured in an ADL. As a result, the component platform can automatically enforce architectural structure on implementation, which helps reduce further architectural erosion. The use of provided and required interfaces as specified by the component model minimizes inflexible, hard-wired dependencies and allows flexible configuration after development time. Considering the scenario of changing the subdomain implementation, this can now be achieved simply by replacing a name in the ADL description (i.e., the SubDomainImpl name in Figure 4). Moreover, the domain component now serves as a reusable unit of functionality that supports the geometric decomposition pattern. Specifically, the component accepts as input the subdomain implementation and the (x,y,z) division and embodies the logic to create and manage the runtime subdomain configuration.

6.2.5 Benchmarks and analysis

6.2.5.1 Comparison between object-based and component-based versions

We first deployed the application on a single cluster so that measurements could be realized in a stable and homogeneous environment, and so that comparison would be possible.

We used a fixed mesh size of 121*121*121. The mesh was sufficiently small so that the application could be deployed on a reduced number of nodes, and sufficiently large so that communication time did not exceed computation time.

The deployment of the Jem3D application proceeds as follows: first, the collector is instantiated on a single node. Second, a set of virtual machines is created according to the deployment descriptor , and using the standard cluster scheduling protocols. Third, active objects are instantiated on the virtual machines. For the component version, once the components are instantiated (as active objects), there are also an assembly and a binding phase to create the system dependencies.

We measured the initialization time as the time between the creation of all remote virtual machines, and the beginning of the computation. We also measured the computation time for a fixed number of iterations with the Jem3d application.

The benchmarks took place on the *azur* cluster in INRIA Sophia-Antipolis, with machines equipped with Opteron processors at 2GHz and 2GB of RAM, and connected through Gigabit Ethernet connections. The JVMs were deployed with an allocated heap size of 1500MB.

The results are presented on figure 6.5.

We observe that:

- The computation times are similar for the component and the object based version, which means that there is *no significant overhead induced by the component framework* during the computation.

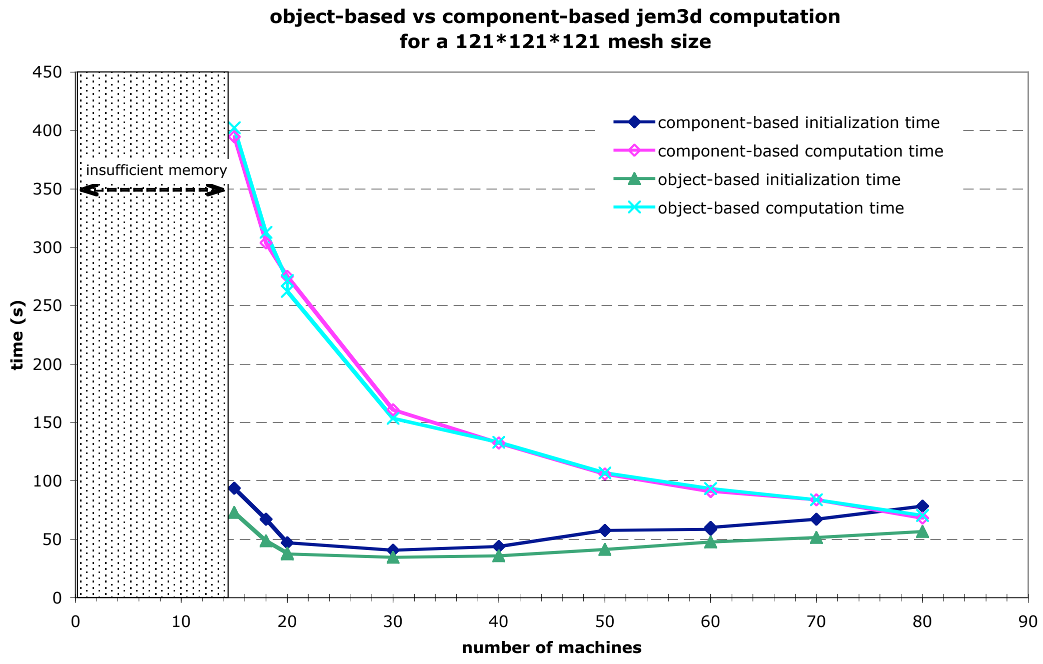


Figure 6.5: *Performance comparison between object based and component based versions of Jem3d*

- The deployment time (referred to as initialization time in the figure) is a little longer for the component-based version. This is due to a more elaborate deployment process, that not only creates component instances, but also assembles them and binds them. *The overhead for deployment seems very much acceptable*.
- With a mesh size of 121*121*121, and with an available heap size of 1500Mo for each computing entity, the computation needs to be distributed on a minimum of 15 machines so that the mesh data can be loaded in memory (the mesh is divided among participants: the higher the number of participants, the smaller the size of the mesh for each participant).

6.2.5.2 Grid scalability

We used the experimental french grid infrastructure Grid'5000 [CAP 05] for performance measurements using several Grid'5000 clusters. The objective was to evaluate the scalability of the component-based version.

We ran several experiments, increasing the mesh size and the number of machines used. One subdomain component or object is deployed on each node. We report the results in table 6.1, also describing the set-up of the different experiments. The parameters which varied were the mesh size and the distribution over the different clusters.

It is important to state that Jem3D does not offer control over the distribution of the computation entities (the subdomains). This problem is only related to the original Jem3D design, not to the component model. All subdomain entities are deployed on a unique virtual node, which is later mapped onto the physical infrastructure. Using several virtual nodes would allow a control over the virtual distribution, hence possibly a

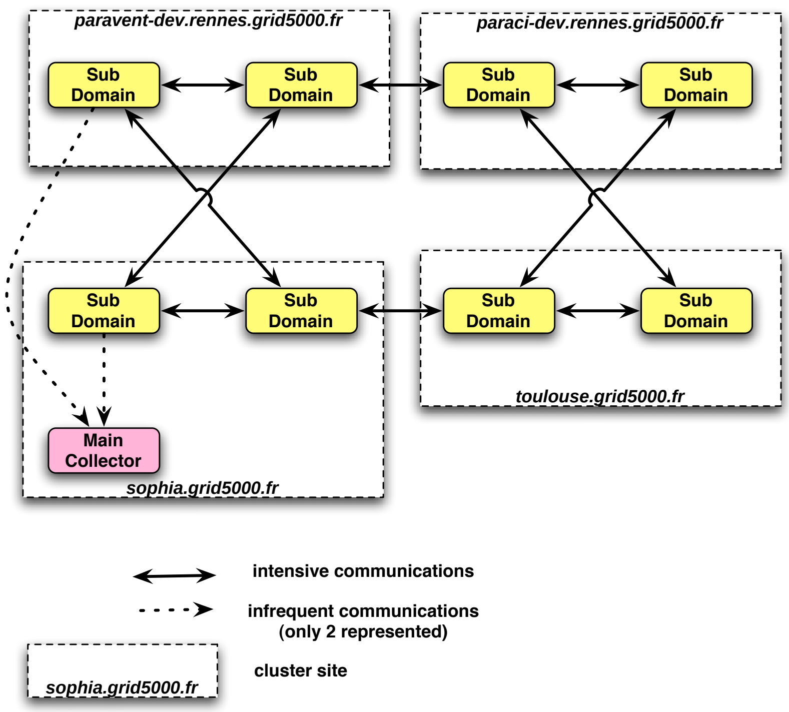


Figure 6.6: *A possible distribution of Jem3D computation over Grid5000 clusters*

control over the physical one, however this was not possible without completely changing the design of Jem3D. As a consequence, some highly communicating neighbors may be located on separate clusters, in which case there is an induced latency overhead in their communications. In the Grid'5000 infrastructure, which uses dedicated and optimized networks (such as RENATER), the latency between machines of a cluster is about 0,05ms, while the latency between clusters can be up to 10ms (this is the case between clusters in Sophia-Antipolis and clusters in Rennes): the latency is up to 200 times higher for inter-clusters than intra-clusters communications.

Measuring computation time in this context is very difficult for a highly coupled application because of both the lack of control over deployment and the inherent instability of Grids. This is why we preferred to present the results from a few experiments, without drawing any conclusion on the performance in this context.

Mesh size	Total Number of Processors Used	Computation time (min)	Processors on Sophia.grid5000.fr Opteron 246 2GHz Linux 2.4.21	Processors on orsay.grid5000.fr Opteron 246 2GHz Linux 2.6.8	Processors on paraci-dev.rennes.grid5000.fr Xeon 2,4 GHz Linux 2.6.11	Processors on parasol-dev.rennes.grid5000.fr Opteron 248 2,2 GHz Linux 2.6.12	Processors on paravent-dev.rennes.grid5000.fr Opteron 248 2,2 GHz Linux 2.6.13	Processors on toulouse.grid5000.fr Opteron 248 2,2 GHz Linux 2.6.10
41*41*41	20	0,46	0	0	0	0	20	0
81*81*81	70	0,94	20	10	20	20	0	0
201*201*201	130	5,15	70	0	60	0	0	0
201*201*201	138	3,85	138	0	0	0	0	0
241*241*241	258	4,29	138	0	120	0	0	0
241*241*241	308	3,72	138	0	120	0	0	50

Table 6.1: *Computational experiments with different Grid configurations*

The results of the experiments as reported in figure 6.1 demonstrate the scalability of the component framework that we developed: we managed to deploy and run a component-based version of the jem3D application on more than 300 processors and up to 4 remote clusters. We were also able to compute with bigger meshes when increasing the number of machines.

6.2.5.3 Discussion

From our experience with the deployment and benchmarking of the Jem3D application, we can draw the following conclusions:

- The component framework does not induce any overhead during computation and the initialization is only slightly longer than for the object-based version. We also demonstrated that the framework is scalable.
- Computational benchmarks for tightly coupled and highly communicating applications need to be performed on homogeneous environments, such as a single cluster. Otherwise performance measurements are unreadable, because inter-cluster communications are several orders of magnitude longer than intra-cluster ones, and because of the inherent instability of Grids: the more different administrative domains involved, the higher the chances of some local disfunction.
- An application, to take advantage of a computational Grid, must provide a *partitioning method at design time*, which at runtime creates partitions depending on the application parameters and the runtime infrastructure. A partition identifies tightly coupled entities which must be colocated, while the coupling between partitions is more loose. Partitions can be attached to virtual nodes, which are mapped on separate deployment infrastructures at deployment time, if needed, resulting in an efficient distribution of the application. Components are a convenient way to design partitioned applications and allow both loosely-coupled applications and tightly-coupled applications. The next section describes how the design of tightly coupled SPMD applications may be facilitated using a component-based approach.
- Grids, by providing large computational infrastructures, *allow new categories of problems to be solved*. For instance, the Jem3D application can solve problems with mesh sizes over 200*200*200, which is impossible on a single cluster with machine equipped of 2GB RAM, because of memory problems.

6.3 Component based SPMD : benchmarks and analysis

6.3.1 Applicative example: Jacobi computation

This example demonstrates and evaluates the capabilities of multicast and gathercast interfaces, as well as the SPMD programming possibilities provided by these interfaces.

For these purposes, we implemented a version of the Jacobi method for solving linear matrix equations. The Jacobi method is an algorithm in linear algebra for determining the solutions of a system of linear matrix equations with largest absolute values in each row and column dominated by the diagonal element. Each diagonal element is solved for, and an approximate value plugged in. The process is then iterated until it converges. The original sequential version is solved by initializing a large matrix, performing a computation on the matrix, then repeating this computation until convergence. It can be easily refactored into a distributed application by partitioning the original matrix. It is a simple example, easily implementable in a SPMD fashion, thanks to the decomposition into sub-matrixes, which compute locally their internal values and swap boundary values with their neighbors, as shown in figure 6.7. Moreover, this program already served as a use case for previous work on OOSPMD programming, which provides us a reference.

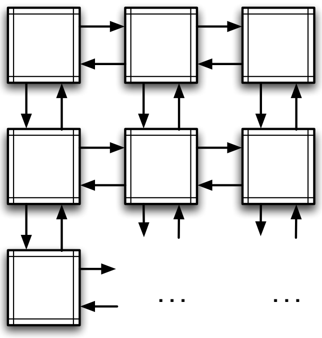


Figure 6.7: *Jacobi computation: matrix elements exchange border data with their neighbors during computation*

For benchmarking purposes, we worked with a fixed number of iterations and fixed matrix values, and we measured the corresponding execution time. We did not wait for convergence.

6.3.2 Design

6.3.2.1 Typing

There is only one type of components in this application: the SubMatrix component. It provides a mandatory gathercast server interface, and has optional multicast and collection interfaces, depending whether borders data is sent using multicast (figure 6.8) or with successive invocations on interfaces of a collection interface (figure 6.8). Border data values are encapsulated as `LineData` objects. The ADL corresponding to this component is the following:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE definition PUBLIC "-//objectweb.org//DTD␣Fractal␣ADL␣2.0//EN" "classpath:
    //org/objectweb/proactive/core/component/adl/xml/proactive.dtd">

<!-- arguments are passed as instantiation parameters-->
<definition name="SubMatrix" arguments="dimensions,coordinates,globalDimensions,
    nbIterations" >

   <interface name="main" role="server" signature="Main" cardinality="multicast"/>
   <interface name="sender" role="client" signature="MulticastDataSender"
       cardinality="multicast" contingency="optional"/>
   <interface name="sender-collection-" role="client" signature="
       CollectionDataSender" cardinality="collection" contingency="optional"/>
   <interface name="receiver" role="server" signature="GathercastDataReceiver"
       cardinality="gathercast"/>

   <!-- this is a primitive component, and here is defined the implementation class
       -->
   <content class="SubMatrixComponent"/>

   <!-- attributes are set right after instantiation of the component-->
   <attributes signature="SubMatrixAttributes">
     <attribute name="dimensions" value="${dimensions}"/>
     <attribute name="coordinates" value="${coordinates}"/>
     <attribute name="globalDimensions" value="${globalDimensions}"/>
     <attribute name="nbIterations" value="${nbIterations}"/>
   </attributes>

   <controller desc="primitive"/>

   <!-- the distribution is abstracted in a virtual node, later mapped onto a
       physical infrastructure using a ProActive deployment descriptor -->
   <virtual-node name="matrixNode"/>

</definition>
```

6.3.2.2 Borders reception and synchronization

The server interface of cardinality gathercast is defined as follows:

```
public interface GathercastDataReceiver {

      public void exchangeData(List<LineData> borders);

}
```

As we can see from the signature of the only method, a list of LineData elements is expected from connected client interfaces. Once one of the clients has sent a LineData element, the gathercast interface will not transfer the invocation until all LineData elements from all connected client interfaces are received. Once all LineData elements are received, they are gathered into a list of LineData elements, and the method defined by GathercastDataReceiver is invoked on the component.

6.3.2.3 Borders sending

Sending border values can be performed in two manners, and this is determined during the binding process: either multicast or collection interfaces are used.

The multicast interface exhibits the following signature:

```
public interface MulticastDataSender {

      @MethodDispatchMetadata(mode = @ParamDispatchMetadata(mode=ParamDispatchMode.
          ONE_TO_ONE))
      public void exchangeData(List<LineData> borders);

}
```

As we can see, a list of LineData elements is given as a parameter to the exchangeData method. This list is built from the values of the borders of the current matrix. The distribution of the LineData elements among the connected server interfaces is specified by the annotation: a *one-to-one* distribution, which means that the server interface of rank i automatically receives the LineData element of rank i in the list. This communication pattern is represented in figure 6.8.

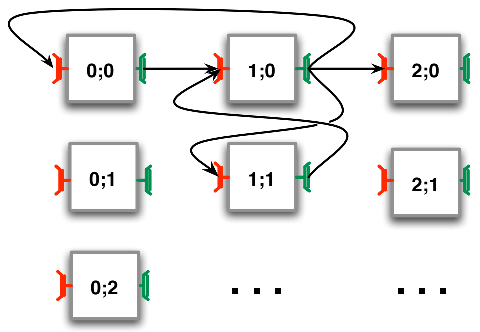


Figure 6.8: *Jacobi computation: borders exchanges through multicast and gathercast interfaces (communications are represented only for component (1;0))*

Another mode for sending borders data is to use a collection of interfaces, where each member of the collection is bound to a neighbor's gathercast interface. In that case, the collection interface has the following signature:

```
public interface CollectionDataSender {

      public void exchangeData(LineData border);

}
```

In this case, the exchangeData method is sequentially invoked on each interface member of the collection interface and sends the corresponding border to the connected neighbor's server interface.

This communication pattern is represented in figure 6.9.

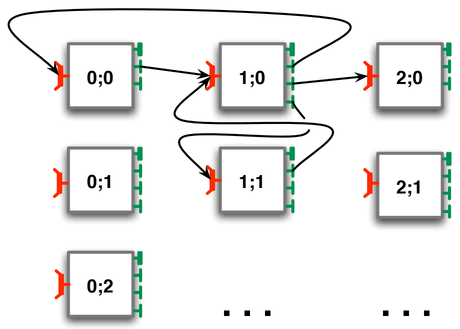


Figure 6.9: *Jacobi computation: borders exchanges through collection and gathercast interfaces (communications are represented only for component (1;0))*

6.3.2.4 Deployment and assembly

As shown in the ADL file above, the SubMatrix components are instantiated on matrixNode virtual nodes, which are mapped onto remote nodes on remote machines in a ProActive deployment descriptor , following the standard deployment model of ProActive.

Once instantiated, the connections between components must be established according to a two-dimensional grid topology, as represented in figure 6.8. This is currently done programmatically, although we plan to provide ADL patterns for a topological automatic assembly.

Binding operations are performed in the following manner, so that the correct 2D grid topology is built; there is a conditional *if* statement on the communication mode, which determines whether bindings are established on collection interfaces or on multicast interfaces.

```
// bind components
for (int x=0; x<nbMatrixX; x++) {
   for (int y=0; y<nbMatrixY; y++) {
   BindingController bc = Fractal.getBindingController(components[x][y]);
   // bindings are performed in the following order : NORTH, EAST, SOUTH, WEST
   if (y!=0) {
     if (useMulticast) {
       bc.bindFc("sender", components[x][y-1].getFcInterface("receiver"));
     } else {
       bc.bindFc("sender-collection-NORTH", components[x][y-1].getFcInterface("
           receiver"));
     }
   }
   if (x!=(nbMatrixX-1)) {
     if (useMulticast) {
       bc.bindFc("sender", components[x+1][y].getFcInterface("receiver"));
     } else {
       bc.bindFc("sender-collection-EAST", components[x+1][y].getFcInterface("
           receiver"));
     }
   }
   if (y!=(nbMatrixY-1)) {
     if (useMulticast) {
       bc.bindFc("sender", components[x][y+1].getFcInterface("receiver"));
     } else {
       bc.bindFc("sender-collection-SOUTH", components[x][y+1].getFcInterface("
           receiver"));
     }
   }
   if (x!=0) {
     if (useMulticast) {
       bc.bindFc("sender", components[x-1][y].getFcInterface("receiver"));
       } else {
         bc.bindFc("sender-collection-WEST", components[x-1][y].getFcInterface("
             receiver"));
       }
   }
  }
}
```

6.3.3 Benchmarks

The benchmarks intend to answer three questions:

- Is a component-based approach usable and scalable for solving SPMD-style problems?

- What is the performance of the component-based approach compared to the OOSPMD-based version?
- What is the performance of multicast communications against sequential one-way invocations through collection interfaces?

The component based version uses the same computation algorithm than the OOSPMD version, the only differences are a different deployment process, and the absence of explicit synchronization barriers.

The fixed parameters in the benchmark are:

- The size for the matrix: 10080*10080 (larger values causing out of memory errors when the computation is only distributed on a small number of machines).
- The values for elements of the matrix: 1000000.
- The values for the (virtual) external border: 0.
- The number of iterations: 1000.
- The deployment was performed on the *azur* cluster in INRIA Sophia, which consists of 105 processors of type AMD Opteron, CPU clock of 2GHz, and 2GB of RAM. The Java Runtime Environment used was in version 1.5.0_04. The deployed Java virtual machines are started with memory parameters `-Xms=1000m -Xmx=1500m`; indeed the JVM has to cope with all the submatrix data which is loaded in memory: when less machines are used, the sizes of submatrixes are higher, therefore more memory is used on each machine.

The parameters which vary are:

- The number of machines used for the computation: the global matrix is decomposed into submatrixes, and the distributed active objects or components each work with a given submatrix.
- The kind of implementation: active objects (OOSPMD), or components.
- When using the component implementation, the communication mode for sending borders data may be through multicast interfaces, or through sequential invocations to members of collection interfaces.

The results of these experiments are represented in figure 6.10. We also represented the performance improvement between the OOSPMD version and the component version using collection interfaces in figure 6.11.

6.3.4 Analysis

We draw three conclusions from this example, regarding respectively the applicability of collective interface, the performance, and the generalization of the SPMD approach.

First, this benchmark is an excellent *proof on concept of collective interfaces* in Fractal components: we managed to deploy and run an SPMD application on up to 100 machines and collective communications were extensively used.

Second, this benchmark demonstrates that our approach for SPMD programming using components and collective interfaces is *valid*, and even *performant compared to the existing OOSPMD-based implementation*. The benchmark results in figure 6.10 show that:

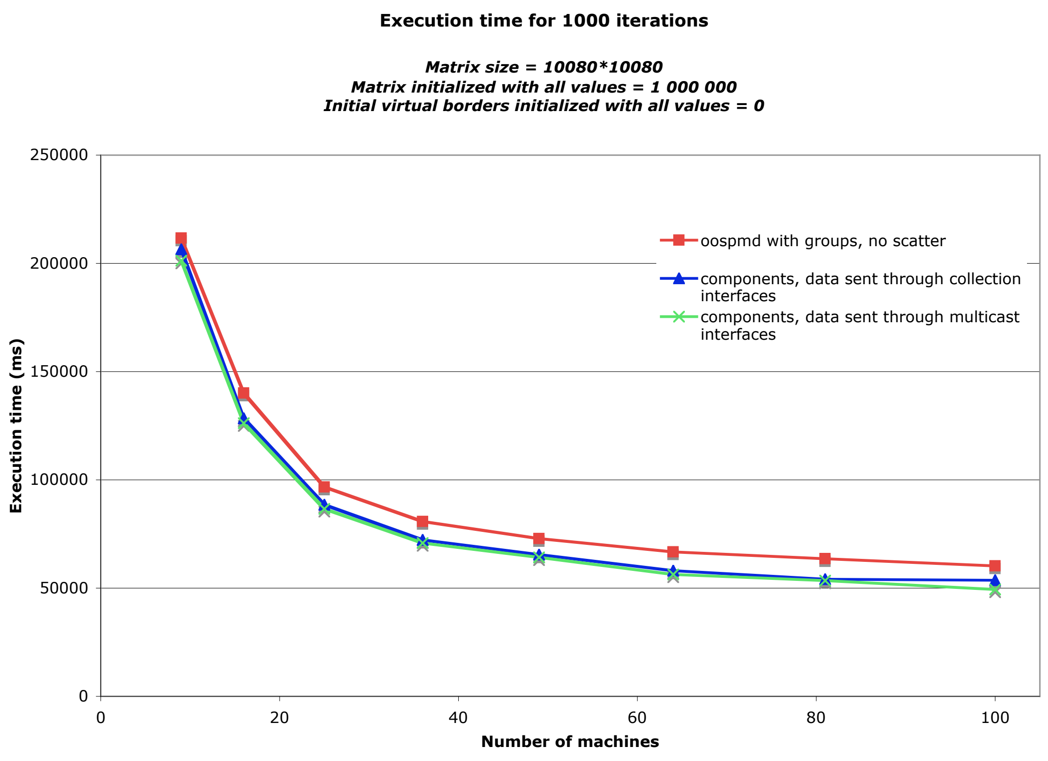


Figure 6.10: *Jacobi computation benchmarks: comparing ProActive OOSPMD, components using multicast and components using collection interfaces for sending data*

- The global computation time decreases as the number of machines involved increases, which is a normal behavior as the computation (whose execution time is non-trivial considering the size of the global matrix) is parallelized. The performance speedup decreases as the number of machines increases because the speedup in pure computation is hindered by the larger number of communications.
- The component based versions are faster than the OOSPMD-based version. We explain this behavior by the explicit synchronization barriers used in the OOSPMD version: setting barriers implies sending extra requests to the SPMD group members; sending and processing requests require a complex and costly reification process which harms performance. On the contrary, there are no explicit barriers in the component version, therefore no extra requests. The process of gathering requests and transforming them into a new invocation is still costly but as stated in the implementation section, there is room for optimization, which means performance can be further improved. We observe a performance gain of about 10% between the OOSPMD version and the component version with collection interfaces, as shown in figure 6.11.
- We observe a slightly better performance when using multicast interfaces than when making successive invocations on members of collection interfaces. We assimilate this gain to the multithreaded mechanisms involved when using multicast interfaces, although further investigation would be required to confirm this, using different data size and distribution strategies.

Third, from a qualitative point of view, this example seems much easier to imple-

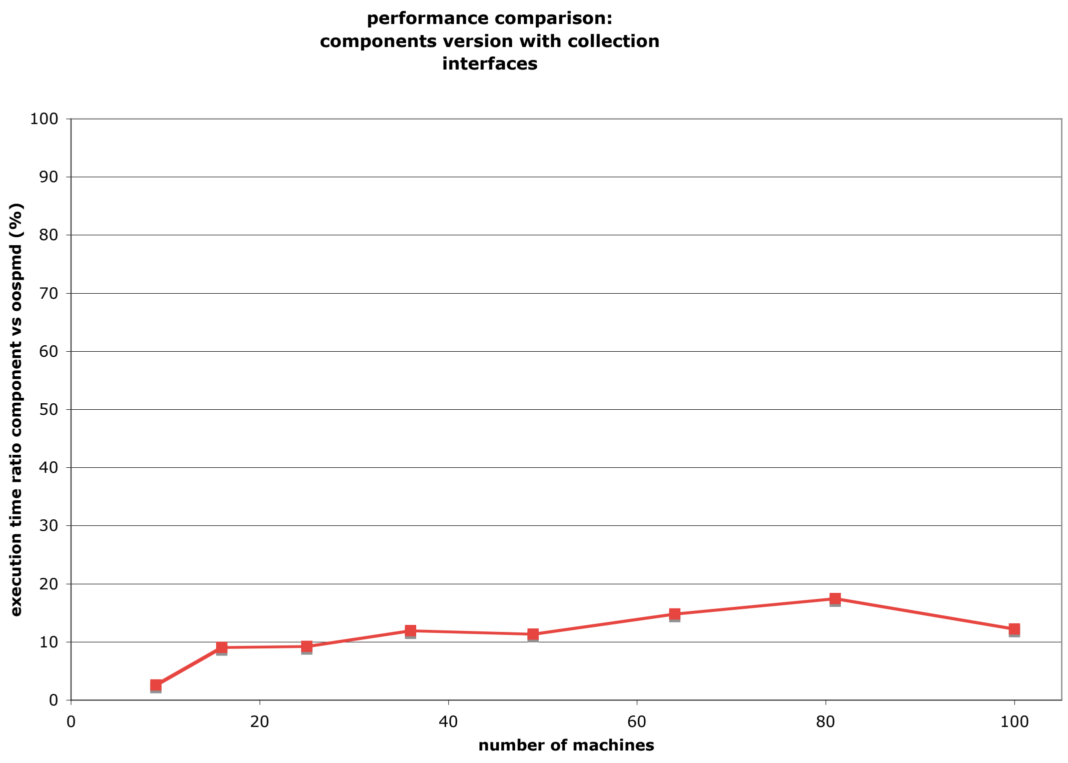


Figure 6.11: *Jacobi computation benchmarks: performance gain of the component based version using collection interfaces compared to the OOSPMD version*

ment than the solution offered by OOSPMD programming: there is no explicit synchronization barrier, and synchronization barriers in OOSPMD are only effective after the end of the current method, which means one has to make sure no communication with neighbors occurs between the barrier statement and the end of the method. Otherwise another synchronization barrier is required.

6.4 Conclusions

The experiments we performed with our component framework highlight the potential of a component-based approach for Grid computing.

From a qualitative point of view, the C3D and Jem3D experiments demonstrated that component designs are clearer and more evolvable than object-based ones.

From a quantitative point of view, the Jem3D and Jacobi experiments demonstrated that the component framework correctly works and that it does not introduce any significant overhead.

In addition, we successfully developed a coupled and highly communicating SPMD application using a novel approach based on collective interfaces, resulting in a complete separation of synchronization concerns from functional ones.

Finally, based on our quantitative experiments, we emphasized the importance of considering distribution at the applicative level, so that resources may be optimally exploited.

Now that we have established the capabilities of our model and framework, we confidently envisage tackling other categories of Grid problems, for which hierarchical component designs are very well suited, such as the NAS Grid benchmarks presented in 3.2.6.1.

Chapter 7

Ongoing Work and Perspectives

7.1 CoreGrid's Grid Component Model

This thesis contributes to the specification and implementation of the Grid Component Model (GCM) defined by the CoreGrid project [COR]. CoreGrid is a european Network of Excellence which gathers academics from 42 institutions in Europe. It aims at strenghtening and advancing scientific and technological excellence in the area of Grid and Peer-to-Peer technologies. It is organized in working groups dealing with: knowledge and data management, programming models, system architecture, Grid information, resource and workflow monitoring services, resource management and scheduling, and tools and environments.

The objective of the working group on programming models is to help reduce the complexity of programming Grid applications, by defining a lightweight component model (the GCM) for the design, implementation and execution of Grid applications. The key problematics addressed by the GCM are programmability, interoperability, code reuse and efficiency.

The GCM adopted Fractal as its base model, and the work of this thesis contributes to the GCM by defining extensions to the Fractal component model with clear semantics for defining collective operations, by proposing a deployment model that takes advantage of virtual nodes to control deployment, and by identifying reconfiguration issues related to the Fractal API.

We intend to complement our work by integrating other aspects of the GCM, in particular those related to autonomicity and interoperability, and we aim at providing an interoperable implementation of the GCM based on the ProActive/Fractal implementation that we presented in this work. This implementation could serve as a reference implementation; it will be named ProActive/GCM and will provide a component framework addressing the needs of Grid programming.

7.2 Pattern-based deployment

Distributed computational applications are designed by defining a functional or domain decomposition, and these decompositions often present structural similarities. We are interested in patterns that contain parameterizable numbers of identical components. The master-workers structure for instance, is commonly used for embarrassingly parallel problems and is implemented in many frameworks such as BOINC, etc.. Bi-dimensional decomposition of a matrix, resulting in two-dimensional grid assemblies of identical computing entities, as shown in 6.3.1, is commonly used. Farm and pipeline are also common structural patterns.

Assembling Grid component applications usually implies assembling a large number of entities, hence performing a large number of assembly operations: remote instantiation, hierarchical composition and bindings. Architecture Description Languages (ADL) offer descriptions of the assembly of component systems and are usually coupled with an ADL processor tool which automatically handles the deployment and assembly of components.

In order to facilitate the design of complex systems with large number of entities, we plan to propose a mechanism for defining *parameterizable assembly patterns* in the Fractal ADL, which is an extensible and parameterizable ADL. The virtual node abstraction will help optimizing the distribution of the applications. Names of interfaces to bind and number of component instances are parameters of the patterns.

There have recently been some proposals in the Fractal community to propose extensions to the Fractal ADL based on AST transformations [FRA06] which would allow this pattern-based configuration. This proves the feasibility of the pattern-based assembly approach, but concrete implementations and examples are yet to be provided, hopefully from our side.

A few examples of configurable patterns are given in figure 7.1, and one should be able to redefine and compose its own configurable patterns.

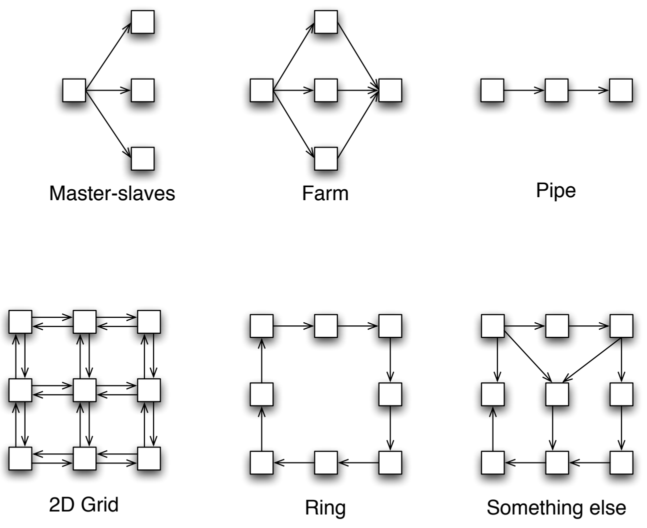


Figure 7.1: *A few assembly patterns*

A typical configuration of a 2DGrid pattern is suggested in the following example, where parameters are: the number of instances and the names of client and server interfaces to be bound. The C component is instantiated `nbInstances` times and neighbor instances establish bindings between the `client` interfaces and server interfaces, according to the predefined `patterns.grid2D` assembly pattern. The 2DGrid is encapsulated in a composite which offers a multicast interface `gridControl` to all enclosed components.

```
<pattern name="grid1" definition="patterns.grid2D(i,␣nbInstances)">
   <interface cardinality="multicast" name="gridControl" signature="Control" role="
      server"/>
   <!-- the following is a ‘‘loop’’ structure that instantiates i components -->
```

```
  <component name="C-%i">
    <interface cardinality="singleton" name="server" signature="Control" role="
        server"/>
    <interface cardinality="multicast" name="client" signature="MulticastClient"
        role="client"/>
    <interface cardinality="gathercast" name="server" signature="MulticastServer"
        role="server"/>
    <content class="CImpl"/>
    <controller desc="primitive"/>
  </component>
 <patternBinding clientItf="client" serverItf="server"/>
 <patternBinding clientItf="this.gridControl" serverItf="control"/>
 </pattern>
```

We note that a generic way to deal with parameterized configurations, without modifying the DTD of the FractalADL was recently proposed on the Fractal mailing list, but complex configuration patterns such as the ones we mentioned probably require an extension of the ADL.

7.3 Graphical tools

Graphical User Interfaces (GUI) are useful for *design and assembly*, and for *monitoring* the runtime activity of component systems.

In an engineering activity as complex as software development, designers must be able to rely on tools for automating as much as possible of the development process, based on software design methods. These tools are usually referred to as Computer Aided Software Engineering (CASE) tools. Some of the most popular CASE tools use the Unified Modeling Language (UML) in order to graphically design software. Indeed, graphical representations allow to rapidly understand the *architecture* of the software, more easily than raw code. They are often used for sketching designs and ideas, but they may also be used as a specification and for an automatic generation of implementation code skeletons; this is the approach advocated by the Model Driven Architecture (MDA) proposal from the OMG for generating Platform Independent Models (PIM).

Components are by nature very suitable for graphical design: they are independent units of software with explicit and strictly defined interfaces. UML 2.0 introduces graphical representation of components, and within the Fractal community, the FractalGUI tool has been proposed for a graphical design of component systems. The FractalGUI is able to generate ADL assembly files from a graphical design, and during this thesis we extended FractalGUI so that it can also represent the composition of virtual nodes (see section 3.3.3.3), and a snapshot is given in figure 7.2.

In the context of Grid computing however, numerous components may be instantiated from parameterized configurations, and it is fundamental to control the geographical composition and the topology of the application. We therefore suggest two directions for enhancing the graphical design of ProActive/GCM applications: the first one is to introduce *abstractions* (graphical patterns) for the design of Grid component systems, and the second one is to *couple graphical representations of component assemblies with distributed monitoring tools* for runtime monitoring.

First, we propose to introduce explicit multiplicity of bindings and components with a star symbol (*), which, as exemplified by figure 6.4, is easily understood. We also suggest to integrate graphical representations of assembly patterns describe in 7.2 within the graphical design.

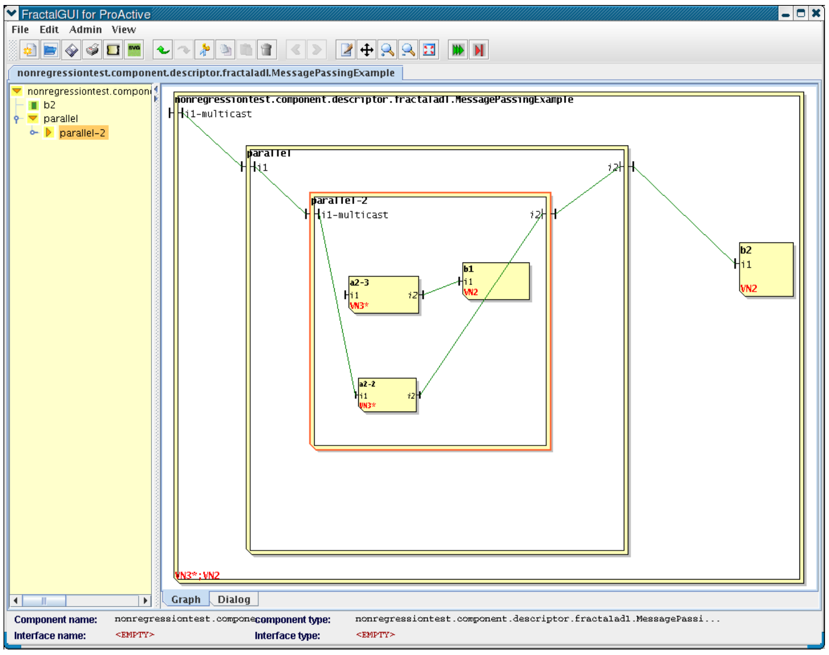


Figure 7.2: *Graphical assembly of components using the FractalGUI*

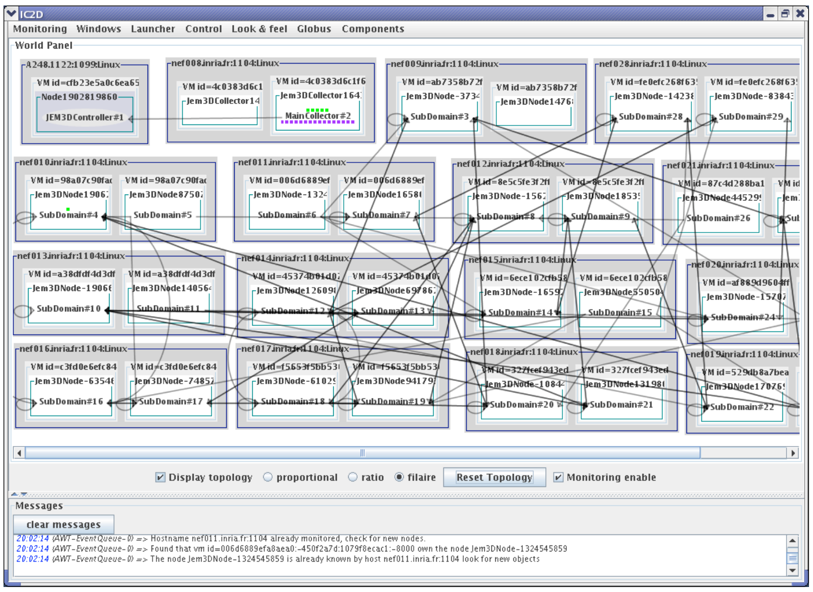
Second, we propose to couple the structural representation of component designs with runtime monitoring capabilities. Several options are envisageable. The Fractal community developed a tool, FractalExplorer, that provides a way for reconfiguring and managing Fractal-based applications at runtime. It however only infers a tree-based representation of the Fractal components. There are also a set of JMX[1] components that can expose Fractal components as JMX MBeans entities, which can be monitored in a standardized way. Finally the FractalGUI itself also proposes some runtime monitoring capabilities, although we did not extend this feature so that it may handle Fractal/ProActive components.

In the context of the ProActive/Fractal framework, we believe a convenient solution would be to couple the extended design capabilities of the Fractal GUI with the runtime monitoring capabilities of the IC2D GUI. IC2D offers a graphical representation of the active objects with two possible views: either a representation of the distributed entities and environments, and the communications between entities (figure 7.3), or a tree-based view for monitoring large number of running jobs. IC2D is currently being refactored with an Eclipse [ECL] plugin architecture, and an extended version of the Fractal GUI could be conveniently integrated. We would therefore couple structural representation with runtime monitoring of the distribution and activity of the components.

The development of the FractalGUI codebase in its current form is now discontinued; a new version has been specified and is currently under development as an Eclipse[2] IDE plugin and for this reason we have not yet integrated the aforementioned proposals in the existing tool. Fortunately, the GridComp project (see section 7.6) defines a working group targeting the development of graphical interface for the design an monitoring of Grid components, and we will push forward the concepts we introduced so that they can

[1]Java Management Extensions (JMX) is a Java technology standard for managing and monitoring applications, system objects, devices (e.g. printers) and service oriented networks. Those resources are represented by objects called MBeans (for Managed Bean)

[2]a popular development environment with a flexible plugin architecture


IC2D
Monitoring Windows Launcher Control Look & feel Globus Components
World Panel
Messages
clear messages

[✓] Display topology ( ) proportional ( ) ratio (●) filaire Reset Topology [✓] Monitoring enable

```
20:02:14 (AWT-EventQueue-0) => Hostname nef011.inria.fr:1104 already monitored, check for new nodes.
20:02:14 (AWT-EventQueue-0) => Found that vm id=006d6889efa8aea0:-450f2a7d:1079f8ecac1:-8000 own the node Jem3DNode-1324545859
20:02:14 (AWT-EventQueue-0) => The node Jem3DNode-1324545859 is already known by host nef011.inria.fr:1104 look for new objects
```

Figure 7.3: *A view of the distribution and communications of a Grid application with the IC2D GUI*

be integrated into the new tool.

7.4 Dispatch and reduction strategies for collective interfaces

In terms of data distribution, we restricted our proposal of collective interfaces (section 4) to data aggregation: either the aggregation of results in a Multicast interface, or the aggregation of parameters for a Gathercast interface. In terms of service invocation, we restricted our proposal to one-to-all and all-to-one invocations ("all" referring to "all connected interfaces"). However, other kinds of distributions are envisageable: collective interfaces may be used for other kinds of data distribution and invocation dispatch. Indeed, according to automatic and user-customizable rules, data could be subject to *reduction*, invocations selected by an election process, and invocation dispatch could be *selective*.

Data reduction corresponds to a transformation of data, according to a given transformation operation, that could be provided by the framework, by the user, or by the data structure itself (through methods of the data objects). For example, MPI defines reduction operations such as `MPI_MAX`, `MPI_MIN`, for determining the minimum and maximum of parameters, or `MPI_SUM` for summing the parameters. However, MPI parameters can only belong to MPI datatypes whereas in our proposal for collective interfaces, parameters may be of any type and could actually define their own reduction operations. Object-defined reduction is useful for inferring reduced types, as in a component model with explicit typed interfaces, data reduction influences the type compatibility of the interface operations. Data reduction is applicable to invocation parameters, for gathercast interfaces, or to results, for multicast interfaces.

In the case of multicast interfaces, selective invocations allow the selection of connected server interfaces that should receive invocations. We note that, in this case, the interface performing the dispatch is more appropriately named *dispatch* interfaces.

The communications may be *one-to-one*[3] , *one-to-some* or *one-to-all*, may be synchronous or asynchronous, and may be performed in parallel. For example, one-to-some communications may be proposed in the following way: the invocation is transformed and forwarded to a subset of the connected server components, similarly to what offers the ICENI dispatch tee (2.4.3.3). The *anycast* [PAR 93] communication mode is another example, realized by delegating the invocation to the most appropriate of the connected components. Other examples include *quorumcast*, which relies on a quorum consensus synchronization, or *probabilistic multicast*, where target components are selected through a probabilistic algorithm.

In the case of gathercast interfaces, selective invocation dispatch corresponds to the selection of invocations, depending on their parameters. Selective dispatch for gathercast interface can be based on *voting* mechanism (quorumcast for instance), with a possible timeout. Voting selection can be customized and parameters such as minimum number of voters, time to vote, minimum percentage etc.. can be specified. For multicast interfaces, selective dispatch correspond to the dispatch of invocations to some of the connected interfaces only, according to various possible selection parameters.

We argue that selective invocations and data reduction mechanisms are valuable for a Grid component model, as they help separating non-functional aspects from functional code, in a customizable manner. Moreover, reduction mechanism already proved useful in MPI frameworks, and voting mechanisms also have pertinent use cases, starting with user inputs in collaborative applications.

Additionally, all-to-all interactions, as proposed in section 4.4 with the notion of gather-multicast interfaces may be further enhanced by integrating selective dispatch (resulting in well-defined some-to-some interactions) and data reduction.

Of course, reduction and selective dispatch have incidences on interface typing, and this must be considered thoroughly.

We therefore plan to clearly specify these extensions to collective interfaces, and propose a reference implementation.

7.5 Adaptivity

One of the challenges of Grid computing is to handle changes in the execution environments, which are not predictable in systems composed of large number of distributed components on heterogeneous environments.

7.5.1 Autonomic computing

The *autonomic computing* paradigm addresses this challenge by building applications out of *self-managed* components. Components which are self-managed are able to monitor their environment and adapt to it by automatically optimizing and reconfiguring themselves. The resulting systems are autonomous and automatically fulfill the needs of the users, but the complexity of adaptation is hidden to them.

The auto-adaptation features can be considered as non-functional features, which may be complex to design and implement. Relying on a component-based design which clearly separates functional and non-functional concerns, and organizes services in a hierarchical manner is therefore a suitable way to design autonomic systems.

[3]A pure dispatch mode implies different signature compatibility conditions than with standard multicast interfaces.

The Bionets project [BIO] is a european project which deals with networks and services in a biologically inspired way. The objective is to provide a fully autonomic environment for networked services, building on a novel approach to information diffusion, communication and filtering that aims at replacing the conventional end-to-end Internet approaches with localized service-driven communications. Bionets will build and experiment with a bio-inspired platform centered around the concept of evolution for the support of autonomic services. Experiments will be performed on a testbed provided by emulating large-scale networks on Grids.

We offer our contribution to the Bionets project by providing a framework for building autonomic components in order to emulate such networks on Grids.

7.5.2 Towards autonomic ProActive/GCM components

Autonomic components are components with encapsulated rules, constraints and mechanisms for self-management and dynamic interactions with other components.

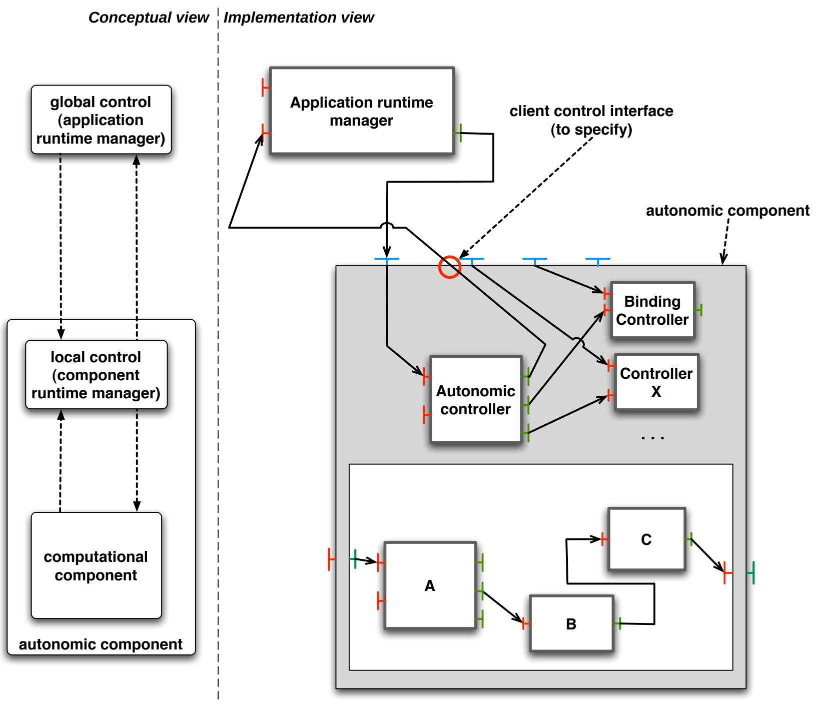


Figure 7.4: *Autonomic components in ProActive / GCM*

We distinguish two kinds of non-functional entities which are involved in autonomic components: coarse-grained entities responsible for global supervision and adaptivity, (Monitoring, Analysis, Planning and Execution, or MAPE, in IBM's view [COR 04]), and lighter-grained entities responsible for individual components. In the ProActive/GCM model, the light-grained entities may be integrated in the membrane of the components, providing *pluggable non-functional services* which can be adapted on-demand for addressing changes in the environment. This approach helps structuring the interactions between non-functional components and it is similar to the micro-components design proposed in [MEN 05], although a notable difference is that micro-components cannot be dynamically modified.

We are currently investigating how the dynamically pluggable non-functional components, acting as controllers in the membrane of ProActive/GCM components, may be implemented as Julia components: we are refactoring the current design of ProActive/GCM to evaluate the potential of this combination of a light-grained implementation of Fractal (Julia) in a coarser-grained implementation (ProActive/Fractal). The targeted design is illustrated in figure 7.4: the local control is handled in the membrane of the components.

The needed collaboration between non-functional services of components also points out the *need for client control interfaces* and controller bindings, which are not yet specified in Fractal or in the GCM.

7.6 Industrial dissemination with the GridComp project

The GridComp project [GC] is a Specific Targeted Research Project supported by the Information Society Technologies of the European Commission. It gathers academics (INRIA, Tsinghua University in Beijing, University of Melbourne among others) and industrials (Atos Origin, IBM, GridSystems) with the main objective of *designing and implementing a component-based framework suitable for the development of efficient Grid applications*. The framework will implement the *invisible Grid* concept, i.e. it will abstract away related implementation details (hardware, OS, authorization and security, load, failure etc...) that usually require considerable programming effort to be dealt with.

The GridCOMP project also has the following objectives:

- Be able to interoperate with existing standards, such as Web Services, WS-RF, Unicore, EGEE gLite.
- Become a "de facto" standard for big industry and SMEs specifying and implementing all the features usually expected from an actual grid programming framework.
- Address both scientific computing and enterprise computing.
- Reach a world wide audience thanks to the involvement of non European partners from South America, Australia and China.

GridCOMP will take the Grid Component Model (to which the models and proposals of this thesis contribute) as a first specification, and use the ProActive implementation (i.e. *the implementation described in this thesis*) as a starting point. Use cases proposed by industrial partners will be implemented, generating valuable feedback on the model and implementation.

Chapter 8

Conclusion

The inherent distributed, complex, and heterogeneous nature of Grids, combined with the willingness to take the best advantage of the resources of Grids, emphasize the need for new programming models that take into account the specificities of Grids.

We proposed a component model and a framework addressing the specificities of Grid computing, particularly the latency, the heterogeneity, the complexity, and the scalability. Our model is based on the Fractal component model and the active object model, augmented with extensions for collective interactions.

Our contributions may be listed as follows:

- Thanks to an analysis of existing programming models for Grid computing, we identified:
  - component-based programming as the most suitable programming model, because it encompasses the other programming models for Grid computing,
  - the shortcomings of existing component models, that are either not designed for parallelism, scalability, or not suitable for heterogeneous environments.
- Proceeding from this study, we verified the adequacy for Grid computing of Fractal, reported as the most general and extensible component model.
- We proposed a component model specific to Grid computing based on the Fractal model and on the active object model, augmented with:
  - a deployment model based on the virtualization of the infrastructure,
  - a specification of collective interfaces, that enable effective multiway communications, parallelism, and synchronization between multiple components.
- We developed an implementation of the component model we defined, based on the ProActive Grid middleware
- We assessed the validity of the proposed component-based approach for programming Grid applications through several experiments. We showed that the component framework does not induce any significant overhead compared to object-based versions of the same application. We also verified the scalability of our framework by deploying these applications on various Grid configurations and more than 300 nodes.
- We proposed a novel approach for SPMD programming with components, where synchronization code is not entangled with functional code.

We did not address all the scenarios of Grid computing in the experiments we performed, and we did not exploit all the features of our component model. In [SNA 03], Grid applications are classified in four categories: loosely-coupled, pipelined, tightly-synchronized, and widely distributed. The applications we experimented belong to the "tightly synchronized" category, and we aim at enlarging the scope of our experiments by developing more diverse applications. Some features of our component model such as customizability, hierarchical structuration, adaptation and dynamicity could therefore be fully exploited.

In order to diversify the scope of possible applications, and to exploit more features of our component model, we plan to implement the NAS-Grid benchmarks, introduced in section 3.2.6.1, which represent reference Grid applications. For these applications, we will take advantage of the hierarchical structuration of our components (figure 3.1), and address loose-coupling through workflow structuration. The GridCOMP project will also bring new scenarios to investigate, particularly in the context of industrial applications.

In conclusion, we successfully extended the Fractal component model in order to address requirements specific to Grid computing, opening many exciting perspectives. A summary of the extensions defined in ProActive/Fractal is given in table 11.1 for the general properties, and in table 11.2 for the Grid-specific properties. We specifically addressed deployment problematics, collective communications, and provided a basis for MxN coupling and interoperability (exposing components as grid services). As participants to the elaboration of the CoreGrid's Grid Component Model (GCM), which is a more comprehensive specification including notably adaptivity, legacy code and interoperability, we aim at providing a reference implementation of the GCM (ProActive/GCM), that will have all required properties described in tables 11.1 and 11.2.

Requirements of component models	Fractal	ProActive/Fractal	ProActive/GCM
Lightweight	✔	✔	✔
Extensible	✔	✔	✔
Hierarchical	✔	✔	✔
Std composition description	✔	✔	✔
Non-functional adaptivity	✔	✔	✔
Reflective	✔	✔	✔
API	✔	✔	✔
Dynamic reconfiguration	✔	✔	✔
Sharing	✔	_[1]	?
Packaging / repository	?	?	✔

✖ no ✔ yes ? Unspecified or unknown

[1] sharing is not yet implemented

Table 8.1: *A comparison of the general features of Fractal, ProActive/Fractal and ProActive/GCM*

Requirements of Grid computing	Fractal	ProActive/Fractal	ProActive/GCM
Deployment framework	?	✔	✔
Multiple deployment protocols	?	✔	✔
Heterogeneity[1]	✔	✔	✔
Interoperability	?	✔	✔
Legacy software	?	?	✔
HPC[2]	-	✔	✔
Collective communications [3]	-	✔	✔
MxN facilities	-	✔	✔
Scalability[4]	-	✔	✔

- no ✔ yes ? Unspecified or unknown

[1] hardware and operating system
[2] targets high performance computin
[3]direct and non broker-based
[4] capability to handle a large number of distributed entities

Table 8.2: *A comparison of how Grid requirements are addressed in Fractal, ProActive/Fractal and ProActive/GCM*

Part II

Résumé étendu en français

(Extended french abstract)

Chapter 9

Introduction

La performance des processeurs croît en suivant la loi de Moore, les capacités de stockage augmentent également de manière exponentielle, l'accès aux ordinateurs et aux équipements électroniques s'est généralisé dans les pays les plus avancés industriellement, et Internet et les technologies sans fil fournissent une interconnexion mondiale et omniprésente. Par conséquent, nous sommes entourés par des capacités croissantes de calcul et de transmission d'information. La quantité et la puissance des ressources disponibles ouvrent de nouvelles opportunités, aux scientifiques pour résoudre de nouvelles catégories de problèmes, aux industriels pour développer de nouveaux modèles économiques, et aux chercheurs en informatique pour étudier comment pouvoir utiliser ces ressources.

Utiliser au mieux ces ressources informatiques implique de les fédérer, ce qui représente une tâche difficile. Le terme Grid computing, ou grilles de calcul, ou encore Grid ou grilles, a été introduit dans les années 90 pour désigner cette tâche, et le Grid computing a été popularisé grâce au livre "The Grid: Blueprint for a New Computing Infrastructure" [FOS 98]. Le concept de Grid a depuis évolué et peut aujourd'hui être défini ainsi:

> Une grille en phase de production est une infrastructure informatique partagée constituée de machines, de logiciels et de bases de données, qui autorise le partage coordonné des ressources et la résolution de problèmes au sein d'organisation dynamiques et multi-institutionnelles, afin de permettre des collaborations scientifiques et industrielles internationales et sophistiquées [LAS 05].

9.1 Problématique

Le Grid computing a des besoins spécifiques inhérents liés à sa nature distribuée et héterogène. Les efforts de recherche ont d'abord porté sur l'accès aux ressources physiques en fournissant les outils pour la recherche, la réservation et l'allocation des ressources, et pour la construction d'organisations virtuelles. L'accès à l'infrastructure de la grille est la première étape nécessaire pour profiter des ressources de la grille, et la seconde étape est d'offrir les modèles et des environnements de développement adéquats. Un nouveau domaine de recherche a donc émergé, centré sur les modèles et outils de programmation permettant de développer efficacement les applications pour les grilles de calcul. Comme mentionné dans [FOS 98], es environnements Grid nécessiteront de repenser les modèles de programmation existants et, probablement, d'inventer de nouveaux modèles plus adaptés aux caractéristiques spécifiques des applications et des environnements Grid.

Les besoins du Grid computing qui doivent être pris en compte par les environnements de programmation peuvent être listés ainsi:

- Distribution: les ressources peuvent être physiquement distantes les unes des autres, avec pour conséquence une latence du réseau et une réduction de la bande passante qui peuvent être significatives et non prévisibles.
- Déploiement: le déploiement d'une application sur une grille est une tâche complexe en termes de configuration et de connexion aux ressources distantes. Les sites utilisés lors du déploiement peuvent être spécifiés à l'avance, ou automatiquement découverts et sélectionnés en fonction de contraintes de déploiement.
- Domaines administratifs multiples: les ressources peuvent être dispersées sur des réseaux différents, chacun avec sa propre politique de sécurité et de gestion des utilisateurs.
- Scalabilité: les applications Grid utilisent un grand nombre de ressources, et le framework doit être en mesure de gérer un grand nombre d'entités, notamment en gérant la latence et en offrant des capacités de parallélisme.
- Hétérogénéité: contrairement au cluster computing (calcul sur grappes), pour lequel les ressources sont homogènes, les grilles de calcul rassemblent des machines provenant de multiples constructeurs, mettant en oeuvre divers systèmes d'exploitation, et utilisant divers protocoles réseaux pour communiquer.
- Interoperabilité et code patrimoine: afin de pouvoir réutiliser des logiciels existants et optimisés, le code patrimoine doit pouvoir être encapsulé, ce qui permet une intégration avancée dans des systèmes plus larges, et l'interopérabilité avec des applications tierces doit être possible.
- Haute performance: les frameworks de programmation doivent être conçus avec des objectifs d'efficacité, notamment en offrant des facilités de programmation d'applications parallèles, car l'un des objectifs du Grid computing est de résoudre des problèmes nécessitant d'importantes puissances de calcul.
- Complexité: les applications Grid peuvent être des logiciels complexes car elles combinent la complexité de briques logicielles très spécialisées avec des problèmes d'intégration, de configuration et d'interopérabilité.
- Dynamicité: une application est susceptible d'évoluer au cours de son exécution (structurellement et fonctionnellement), à cause de changements environnementaux (afin de garantir une performance optimale, ou dans le cas de dysfonctionnements), ou par l'addition ou la suppression de ressources alors que l'application est en cours d'exécution (une grille est intrinsèquement dynamique).

9.2 Objectifs et contributions

Ce travail appartient au second domaine de recherche précédemment cité, à savoir, l'étude de modèle et outils de programmation pour le développement et l'exécution d'applications pour les grilles. Notre principal objectif est de définir un modèle de programmation et un framework d'exécution qui réponde aux besoins du Grid computing.

Nous considérons que certains des besoins du Grid computing doivent être pris en charge par la couche middleware, comme la distribution, la gestion de domaines administratifs multiples et l'interoperabilité. Les autres besoins doivent être pris en charge par le modèle de programmation. Dans le modèle de programmation que nous proposons, nous cherchons à simplifier la conception des applications Grid et nous cherchons à founir des abstractions de haut niveau pour la conception des interactions entre multiples entités distribuées.

Notre approche est basée sur un modèle de composants hiérachique reposant sur le concept d'objets actifs et complémenté par des communications collectives.

Les principales contributions de cette thèse sont:

- une analyse de l'adéquation des modèles de programmation existants, et en particulier des modèles de programmation par composants, en regard des besoins du Grid computing,
- une proposition d'un modèle de composants destiné à répondre spécifiquement à ces besoins, grâce à la composition hiérarchique, et à un modèle de déploiement adapté au Grid computing,
- une specification d'interfaces collectives, en tant que moyen pour concevoir des interactions multiples entre composants distribués, le parallélisme et la synchronisation entre composants
- un framework de composants qui implante le modèle de composants proposé et la spécification des interfaces collectives.

9.3 Plan

Ce document est organisé de la manière suivante:

- Dans le chapitre 2, nous situons notre travail dans le contexte de l'architecture logicielle du Grid computing. Nous présentons les modèles de programmation existant pour le Grid computing afin de justifier notre approche, et nous évaluons les différentes possibilités permettant des communications collectives. Nous exposons comment les besoins du Grid computing sont traités dans les modèles et frameworks existants; cela nous permet de mettre en évidence ce qui doit être amélioré. Enfin, nous presentons le modèle des objets actifs et le middleware ProActive qui servent de base à notre travail.
- Dans le chapitre 3, nous proposons un modèle de composants pour le Grid computing qui étend le modèle de composants Fractal.
- Dans le chapitre 4, nous ajoutons à notre modèle de composants une spécification d'interfaces collectives, en tant que moyen de gérer les interactions multiples au niveau des interfaces des composants. Nous montrons comment les interfaces collectives peuvent être avantageusement utilisées pour des applications couplées.
- Le chapitre 5 décrit notre framework de composants qui implante le modèle de composants spécifié aux chapitres 3 et 4.
- Le chapitre 6 rapporte les expériences réalisées à l'aide de notre framework de composants: nous démontrons son intérêt pour le Grid computing, sa capacité de passage à l'échelle, et nous montrons comment il peut être utilisé pour des applications SPMD. Nous présentons également une méthodologie pour passer d'une

application conçue avec des objets distribués, vers une application conçue avec des composants distribués, et nous soulignons certains problèmes liés au déploiement d'applications couplées dans un environnement Grid.

- Au chapitre 7 nous proposons un aperçu de nos travaux en cours et des améliorations que nous envisageons, et nous soulignons le potentiel de notre model dans les contextes académiques et industriels.
- Le chapitre 8 conclue et résume les principales réalisations de cette thèse.

Chapter 10

Résumé

10.1 Etat de l'art

Dans ce chapitre, nous justifions notre choix d'un modèle de composants hiérarchique basé sur le concept d'objets actifs et étendu par des communications collectives, en évaluant les modèles de programmation existants pour le Grid computing, en évaluant les approches possibles pour offrir des communications collectives, et en mettant en évidence quels besoins ne sont pas pris en charge par les modèles et frameworks existants.

10.1.1 Positionnement

L'un des objectifs fondamentaux des grilles est de fournir un accès aisé voire transparent à des ressources de calcul hétérogènes et réparties sur des domaines administratifs différents. Ceci est également dénommé "virtualisation des ressources". Notre objectif est d'offrir à la fois un environnement d'exécution réalisant la virtualisation, et un modèle de programmation permettant de développer de manière optimale en vue de déployer sur l'environnement d'exécution.

Une représentation classique de l'architecture logicielle des grilles est une représentation en quatre couches.

- La couche la plus basse (Grid fabric) permet la virtualisation des ressources en fournissant un accès aux ressources et aux systèmes par l'intermédiaire de schedulers pour clusters, de protocoles réseaux et de systèmes d'exploitation.
- Au dessus, la couche Grid middleware englobe l'ensemble des services permettant de gérer les ressources fédérées et la manière de les partager au sein d'une grille.
- La troisième couche fournit les outils et les modèles qui permettent de programmer les applications pour les grilles.
- Enfin, la couche la plus haute rassemble portails d'applications, Problem Solving Environments et applications clientes de la grille en général.

Notre travail contribue à la troisième couche, modèles et outils pour la grille, car nous proposons à la fois un modèle de programmation et un framework permettant de déployer et exécuter les applications Grid.

Considérons maintenant le positionnement de notre travail au sein des modèles de programmation. Divers modèles de programmation ont été étudié et mis en oeuvre afin de faciliter le développement d'applications pour les grilles. Nous avons noté que la plupart présentent des avantages pour certains types d'applications, mais qu'en général ils ne permettent pas de programmer efficacement l'ensemble des applications envisageables pour les grilles.

Parmi ces modèles, les architectures à services se distinguent par leur capacité à gérer des applications à gros grain et faiblement couplées, et leur nature interopérable, au détriment néanmoins de la performance.

Les modèles à passage de messages (extensions de MPI utilisant des middlewares de Grid pour l'accès aux ressources virtualisées) permettent de s'affranchir des contraintes liées à la latence, offrent de bonnes performances pour les communications et proposent des opérations collectives, cependant ils n'offrent pas les abstractions nécessaires au Grid computing, en particulier l'hétérogénéité, l'interopérabilité, l'adaptativité ou la dynamicité.

Les modèles à squelettes, comme ASSIST ou HOC-SA, permettent de concevoir des applications clairement et efficacement structurées, peuvent permettre d'optimiser le parallélisme, et sont interfaçables avec infrastructures de grilles grâce aux middlewares comme Globus. L'utilisation des squelettes reste cependant une facilité algorithmique et n'est pas forcément idéale dans le cadre d'applications dynamiques et faiblement couplées.

Les modèles de type "serveur en réseau" (Network-Enabled Servers) et les modèles GridRPC desquels ils proviennet conviennent à des traitements par lots, pour lesquels les tâches de calcul sont isolées et fournie à une entité "serveur", qui se charge de distribuer les lots de calcul sur l'infrastructure de manière optimale.

Les modèles à objets distribués sont également utilisés et bénéficient de leur popularité dans le monde industriel, mais imposent une spécification explicite des dépendances entre objets applicatifs, et en général manque de modularité, de description externe des systèmes et de facilitités de paquetage.

Finalement, nous considérons que les modèles à composants sont les plus à même de répondre à l'ensemble des besoins des grilles, notamment l'utilisabilité, car ils sont plus généraux et peuvent encapsuler les fonctionnalités offertes par les autres modèles.

10.1.2 Modèles à composants pour le Grid computing

Les modèles de programmation par composants sont largement répandus dans le monde industriel et académique. Ils proposent un niveau d'abstraction supérieur à celui des modèles objets et reposent sur les trois principes fondamentaux d'encapsulation (boîte noire), de composition (entités encapsulée connectables) et de description (description du logiciel séparée du code applicatif).

Pour le Grid computing, les modèles de composants les plus communs ne sont pas adaptés, car ils ne prennent pas en compte les notions de parallélisme et de distribution à large échelle. Par conséquent, de nouveaux modèles de composants ont été étudiés afin de bénéficier des avantages des modèles à composants dans le cadre du Grid computing. Nous décrivons notamment CCA, Iceni et GridCCM.

10.1.3 Communications collectives

Les communications collectives sont un outil fondamental pour gérér le parallélisme, la synchronisation entre un grand nombre d'entités distribuées, et le couplage de code. Nous présentons les stratégies et mécanismes classiques de communications collectives pour ces types de problèmes en excluant les mécanismes de publication.

En conclusion de notre étude sur les modèles de programmation pour la grille, nous présentons une classification des modèles de composants en considérant à la fois leurs propriétés générales et leur capacité à répondre aux besoins des grilles, et nous montrons que Fractal est le candidat le plus prometteur.

10.1.4 Contexte: objets actifs avec ProActive

Ce travail constitue une extension de la bibliothèque ProActive, une bibliothèque de programmation et de déploiement d'applications Grid. Le modèle de programmation de ProActive repose sur la notion d'objets actifs, qui sont des entités indépendantes, distribuables, monothreadées, et qui interagissent de manière asynchrone. ProActive est un logiciel proposant de nombreuses facitilités spécifiques aux grilles et grâce à son architecture extensible, représente une base riche pour implanter un framework de composants.

10.2 Un modèle à composants pour le Grid computing

10.2.1 Adéquation du modèle Fractal pour le Grid computing

Afin de répondre aux besoins du Grid computing, décrits dans l'introduction, nous vérifions l'adéquation avec les besoins de la grille du modèle Fractal, qui est le modèle de composants que nous avons distingué comme le plus général et le plus extensible parmi ceux étudiés.

Nous remarquons que certains aspects du modèle Fractal doivent être étendus afin de répondre aux besoins des grilles, en particulier concernant le déploiement et les communications collectives.

10.2.2 Un modèle à composants basé sur le modèle d'objets actifs

Nous proposons donc un modèle de composants basé sur le modèle Fractal et sur le modèle d'objets actifs, ce qui permet de bénéficier des avantages des deux modèles ainsi que d'une bibliothèque de programmation existante et extensible (ProActive).

Dans ce modèle, un composant est un objet actif et peut être distribué, et les communications entre composants sont asynchrones. Le déploiement est basé sur une virtualisation de l'infrastructure sous forme de noeuds virtuels, et la description d'un système de composants peut prendre avantage de cette virtualisation afin de contrôler la coallocation éventuelle de composants couplés.

Certains problèmes rencontrés concernant la dynamicité des configurations sont isolés et étudiés; ils sont liés à la combinaison de compositions hiérarchiques et de composants actifs et monothreadés.

10.3 Interfaces collectives

Les interfaces collectives permettent de contrôler les interactions multiples entre composants, c'est à dire 1 à n, n à 1 ou n à m, en particulier le parallélisme et la synchronisation.

10.3.1 Motivations

Nous montrons d'abord que les cardinalités actuelles des interfaces dans le modèle Fractal ne permettent pas les communications 1 vers n ou n vers 1 sans l'utilisation de composants intermédiaires, et nous considérons, pour des raisons de simplicité, d'utilisabilité et d'encapsulation, que la sémantique des communications collectives doit être specifiée au niveau des interfaces.

10.3.1.1 Proposition

Nous proposons une spécification d'interfaces collectives pour le modèle Fractal, et nous presentons deux types d'interfaces: multicast et gathercast.

Les interfaces multicast permettent de gérer les interactions de type 1 vers n. Une interface multicast transforme une invocation en une liste d'invocations. La distribution des paramètres des méthodes est configurable et s'effectue soit par copie simple, soit par extraction et distribution des parametres contenus dans une liste, auquel cas la signature des méthodes varie entre l'interface multicast et les interfaces serveuses connectées.

Les interfaces gathercast permettent de gérer les interactions de type n vers 1. Une interface gathercast transforme une liste d'invocation en une seule invocation. Les paramètres des méthodes sont par défaut aggrégés au sein d'une liste, et peuvent éventuellement faire l'objet d'une réduction ou d'une sélection. La signature des méthodes varie entre une interface gathercast et les interfaces clientes qui lui sont connectées.

Si le language utilisé le permet, comme Java par exemple, il est possible de préciser la configuration des interfaces collectives à l'aide d'annotations.

10.3.1.2 Couplage d'applications

Un domaine d'application des interfaces gathercast et multicast est le couplage d'applications. Le couplage d'applications prend avantage des propriétés de synchronisation et de parallélisme, et de la séparation entre la logique de synchronisation et de parallélisme, et la logique applicative.

Deux types d'applications sont concernées par les capacités de couplage offertes par les interfaces collectives: la programmation SPMD (Programme Unique Données Multiples) et le couplage de composants parallèles.

La programmation SPMD peut être appliquée aux composants en considérant des instances du même composant comme des programmes identiques, et en définissant les interactions multiples entre ces composants par l'intermédiaire d'interfaces gathercast et multicast. Le résultat est analogue à l'utilisation des communications collectives en MPI, mais dans notre cas, il n'y a pas d'intrusion dans le code applicatif.

Le problème MxN désigne le problème de communiquer et échanger des données entre des programmes parallèles, depuis un programme parallèle contenant M processus, vers un programme contenant N processus. Les programmes parallèles peuvent être conçus à l'aide de composants composants de même types, et ces composants peuvent être encapsulés au sein d'un composant composite, que l'on peut désigner comme composant "parallèle". Les interfaces multicast et gathercast permettent de gérer la parallélisation des accès aux composants au niveau du composite englobant. Le couplage entre composants parallèles revient donc à effectuer de la manière la plus efficace la liaison entre deux composites "parallèles". Nous pensons qu'une implantation optimale et automatisable de cette liaison peut être réalisée à l'aide des interfaces multicast et gathercast, localement au niveau des composants similaires contenus dans les composites "parallèles".

10.4 Un framework à composants basé sur les objets actifs

Afin de proposer une plate-forme d'expérimentation, de validation, et de mise en application de nos concepts, nous avons implanté le modèle de composants proposé, sous la forme d'une extension de la bibliothèque ProActive.

Nous avons réalisé cette implantation avec comme objectifs:

- De baser l'implantation sur le concept d'objets actifs déjà offert par ProActive.
- D'enrichir la bibliothèque ProActive en offrant un nouveau modèle de programmation par composants, compatible avec les fonctionnalités classiques de la bibliothèque.
- De fournir un framework qui soit réponde aux besoins du Grid computing, en combinant les fonctionnalités de distribution et de déploiement de ProActive, avec les propriétés offertes par notre modèle de composants.
- D'offrir un framework adaptable et configurable.
- D'implanter notre spécification d'interfaces collectives.

L'architecture de notre framework à composants repose sur l'architecture à méta-objets de ProActive. Nous avons ajouté un ensemble de méta-objets permettant de gérer les aspects composants, et nous avons adapté la représentation des objets actifs afin de pouvoir représenter des instances de composants, et non simplement des instances de classes. Les contrôleurs, permettant de gérer les aspects non-fonctionnels des composants, et d'intercepter les invocations, sont entièrement configurables et nous proposons un ensemble minimum de contrôleurs. La gestion du cycle de vie des composants est également configurable et est compatible avec la redéfinition de l'activité des objets actifs.

Nous avons également implanté les notions d'interfaces collectives introduits dans le chapitre 4. La configuration des interfaces collectives s'effectue par le biais d'annotations Java au niveau des interfaces, et la vérification de la compatibilité de typage des liaisons s'effectue à l'exécution, lors du processus de liaison.

10.5 Méthodologie et benchmarks

Ce chapitre rapporte les expériences que nous avons réalisées afin d'évaluer le fonctionnement, l'utilisabilité et les performances de notre modèle et de son implantation.

Nous décrivons trois expériences principales: C3D est une simple application collaborative existante refactorée grâce aux composants, Jem3D est une application de calcul numérique déployable sur grilles, que nous avons refactorée sous forme de composants, et nous avons également développé une version de la méthode de Jacobi pour la résolution de systèmes linéaires, sous formes de composants SPMD reliés par des interfaces collectives.

L'application C3D nous a permis de valider notre approche, de vérifier son applicabilité pour des applications distribuées et relativement simple, et d'obtenir une première évaluation du travail requis pour un passage d'objets à composants.

L'application Jem3D nous a permis de définir un méthodologie de transformation d'un code objet en un code composants, ce qui est utile pour pouvoir apporter facilement les avantages d'une structuration en composants à des codes objets existants. Nous avons également pu vérifier avec Jem3D la capacité de passage à l'échelle de notre framework (déploiement sur plus de 300 noeuds et 4 clusters) et nous avons mesuré que le framework de composants n'induit pas de baisse de performance significative par rapport à une application objets.

L'application Jacobi est l'implantation d'une méthode de résolution de systèmes linéaires matriciels. Le calcul peut être facilement découpé et les entités de calcul peuvent être implantées sous formes de composants échangeant des résultats de calcul avec leur voisins. Cette expérience nous a permis de mettre en oeuvre les interfaces gathercast et multicast et de comparer le résultat avec une version existante sous forme d'objets SPMD. Les interactions collectives sont définies par l'assemblage des interfaces collectives des composants. Nous avons pu vérifier le comportement normal du calcul et les performances intéressantes. Nous en concluons que méthode de calcul SPMD utilisant les interfaces collective est une méthode efficace et non-intrusive.

10.6 Travaux en cours et perspectives

10.6.1 GCM, le modèle de composants de CoreGrid's

Notre travail contribue à l'élaboration du modèle de programmation par composants pour la grille proposé par le réseau d'excellence académique CoreGrid, le GCM (Modèle de Composants pour la Grille). Le GCM a adopté Fractal comme modèle de base et les travaux de cette thèse contribuent au GCM par la définition d'extensions au modèle Fractal, en particulier la specification d'interfaces collectives ayant une sémantique claire, en proposant un modèle de déploiement prenant avantage de la virtualisation de l'infrastructure pour contrôler le déploiement, et en identifiant des difficultés de reconfiguration liées à l'API Fractal.

10.6.2 Patrons de déploiement

Nous proposons de faciliter l'assemblage et le déploiement d'applications Grid par la définition de patrons d'assemblages configurables et composables, qui permettront de créer facilement des systèmes à base de composants suivant des patrons de configurations communément utilisés. Pipe, master-slaves, grilles 2D et 3D sont des exemples classiques de tels patrons.

10.6.3 Outils graphiques

Les outils graphique sont utiles pour la conception, l'assemblage et le contrôle à l'exécution des systèmes de composants. Au même titre que les environnements de dévelopement pour les languages classiques, les outils graphique permettent de fournir des outils de haut niveau pour la définition et l'analyse de systèmes de composants. De plus, dans le cadre des grilles, les outils de contrôle à l'exécution permettent de vérifier le bon comportement d'une application distribuée.

La spécification statique d'applications composants à l'aide de descripteurs d'architectures (ADL) permet de proposer facilement des outils graphiques pour concevoir et manipuler ces assemblages. Dans le cadre des grilles, nous remarquons cependant que de nouvelles abstractions doivent être introduites afin de prendre en compte le nombre d'instances potentiellement important, et de fournir non seulement une représentation structurelle, mais également une représentation distribuée des systèmes à composants.

10.6.4 Dispatch et réduction pour les interfaces collectives

Notre implantation actuelle des interfaces collectives n'offre pas l'ensemble des fonctionalités présente dans notre spécification, en particulier concernant la réduction et la sélection de paramètres et de résultats. Nous souhaitons donc étendre notre framework afin de proposer égalemetn ces fonctionnalités.

10.6.5 Adaptativité

L'un des challenges du Grid computing est de gérer les changements dynamiques de l'environnement d'exécution, liés à la charge de la machine, au réseau, ou encore, à la restructuration dynamique des applications.

Une approche pour répondre à ce challenge est l'informatique autonomique (autonomic computing), qui propose de rendre auto-adaptables les entités logicielles. Dans le cadre du projet européen Bionets, notre équipe délivrera un plate-forme de simulation sur grille capable de gérer des composants autonomiques. Pour cela, nous étudions des techniques de composition des aspects non-fonctionnels intra et inter composants, notamment par l'utilisation d'un framework de composants léger au sein de notre framework de composants à plus gros grain.

10.6.6 Dissémination industrielle

Le projet GridComp est un projet européen appuyé par la commission européenne. Il rassemble des académiques (INRIA, Tsinghua University à Beijing, University of Melbourne entre autres) et des industriels (IBM, Atos Origin, GridSystems), dans l'objectif de concevoir et d'implanter un framework standardisé de composants pour le développement d'applications Grid. GridComp utilisera le GCM proposé par CoreGrid comme première spécification et ProActive/Fractal (que nous avons développé au cours de cette thèse) comme support d'exécution, et les cas d'utilisation proposés par les partenaires industriels permettront d'affiner le modèle et l'implantation ProActive/Fractal.

Chapter 11

Conclusion

La nature distribuée, complexe et hétérogène des grilles, et la volonté de bénéficier des ressources des grilles, mettent en évidence le besoin de nouveaux modèles de programmation qui prennent en compte les besoins des grilles.

Nous proposons un modèle de programmation par composants et un framework qui prennent en considération les spécificité du Grid computing, en particulier la latence, l'hétérogénéité, la complexité, et le passage à l'échelle. Notre modèle est basé sur le modèle d'objets actifs et sur le modèle de composants Fractal, étendu pour gérer les interactions collectives.

Nos contributions peuvent être résumées de la manière suivante:

- Grâce à une analyse des modèles de programmation existants pour le Grid computing, nous avons identifié:
  - la programmation par composants comme le paradigme le plus adapté, car il englobe les autres paradigmes de programmation
  - les limitations des modèles de composants existants, qui ne sont: pas conçus pour le parallélisme, ou pour le passage à l'échelle, ou pour des environnements hétérogènes.
- A partir de cette étude, nous avons vérifié l'adéquation de Fractal pour le Grid computing, notre étude mettant en évidence Fractal comme le modèle le plus général et le plus extensible.
- Nous proposons un modèle de composants spécifique au Grid computing basé sur le modèle Fractal et sur le modèle d'objets actifs, étendus par:
  - un modèle de déploiement basé sur une virtualisation de l'infrastructure,
  - une spécification des interfaces collectives, qui permet : des communications multi-entités efficaces, le parallélisme, et la synchronisation entre multiples composants.
- Nous avons développé une implantation du modèle de composants que nous avons défini. Cette implantation est basée sur ProActive, un middleware de Grid.
- Nous avons vérifié, par plusieurs expériences, la validité de l'approche par composants proposée pour la programmation d'applications Grid. Nous avons montré que le framework de composants n'introduit pas de baisse de performance significative, en comparaison avec les versions "objets" des mêmes applications. Nous avons également vérifié la capacité de passage à l'échelle de notre framework en

déployant ces applications sur diverses configurations de grilles et sur plus de 300 noeuds de calcul.

- Nous avons proposé une nouvelle approche pour la programmation SPMD par composants, dans laquelle le code de synchronisation n'est pas imbriqué dans le code fonctionnel de l'application.

Nous n'avons pas pris en compte l'intégralité des scenarios possibles de Grid computing dans nos expériences, et nous n'avons pas exploité toutes les fonctionnalités de notre modèle de composants. Dans [SNA 03], les applications Grid sont classées en quatre catégories: faiblement couplées, pipelinées, fortement couplées, et largement distribuées. Les applications que nous avons utilisées appartiennent à la catégorie "fortement couplées", et nous souhaitons élargir le domaine de nos expériences en développant des applications plus variées. Certaines des fonctionnalités de notre modèle de composants n'ont pas été entièrement exploitées alors que notre framework implante ces fonctionnalités, telles la customisation, la structuration hiérarchique, l'adaptation et la dynamicité.

Afin de diversifier le champ de nos applications potentielles, et afin d'exploiter l'ensemble des fonctionnalités de notre modèle de composants, nous prévoyons d'implanter les NAS-Grid benchmarks, présentés en section 3.2.6.1 et qui constituent des applications Grid de référence. Pour ces applications, nous mettrons à profit la structuration hiérarchique de nos composants (figure 3.1), et nous prendrons en charge le couplage faible grâce à une structuration en workflow.

En conclusion, nous avons étendu avec succès le modèle de composants Fractal afin de prendre en charge les besoins spécifiques du Grid computing, ce qui nous permet d'entrevoir un grand nombre de perspectives passionnantes. Un résumé des extensions définies dans le framework ProActive/Fractal est fourni en table 11.1 pour les propriétés générales, et en table 11.2 pour les propriétés spécifiques à la grille. Nous avons spécifiquement considéré les problématiques de déploiement, de communications collectives, et nous avons fourni une base pour le couplage MxN et pour l'interoperabilité (en exposant les composants en tant que services web). En tant que participants à l'élaboration due modèle de composants de CoreGrid (le GCM, qui est une spécification plus exhaustive incluant notamment l'adaptativité, la gestion de code patrimoine et l'interopérabilité complète), nous souhaitons fournir une implantation de référence du GCM (ProActive/GCM), qui présentera l'ensemble des propriétés décrites tables 11.1 et 11.2.

Requirements of component models	Fractal	ProActive/Fractal	ProActive/GCM
Lightweight	✔	✔	✔
Extensible	✔	✔	✔
Hierarchical	✔	✔	✔
Std composition description	✔	✔	✔
Non-functional adaptivity	✔	✔	✔
Reflective	✔	✔	✔
API	✔	✔	✔
Dynamic reconfiguration	✔	✔	✔
Sharing	✔	–[1]	?
Packaging / repository	?	?	✔

✖ no ✔ yes ? Unspecified or unknown

[1] sharing is not yet implemented

Table 11.1: *Compaison des propriétés générales entre Fractal, ProActive/Fractal et ProActive/GCM*

Requirements of Grid computing	Fractal	ProActive/Fractal	ProActive/GCM
Deployment framework	?	✔	✔
Multiple deployment protocols	?	✔	✔
Heterogeneity[1]	✔	✔	✔
Interoperability	?	✔	✔
Legacy software	?	?	✔
HPC[2]	–	✔	✔
Collective communications [3]	–	✔	✔
MxN facilities	–	✔	✔
Scalability[4]	–	✔	✔

– no ✔ yes ? Unspecified or unknown

[1] hardware and operating system
[2] targets high performance computin
[3]direct and non broker-based
[4] capability to handle a large number of distributed entities

Table 11.2: *Comparaison de la prise en charge des besoins des grilles entre Fractal, ProActive/Fractal et ProActive/GCM*

Bibliography

[ABR 95] DAVID ABRAMSON, ROK SOSIC, J. GIDDY, and B. HALL. "Nimrod: a tool for performing parametrised simulations using distributed workstations". In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*, 1995.

[ALD 06] MARCO ALDINUCCI, MASSIMO COPPOLA, MARCO DANELUTTO, MARCO VANNESCHI, and CORRADO ZOCCOLO. "*Grid Computing: Software environments and Tools*", chapter ASSIST as a research framework for high-performance Grid programming environments. Springer Verlag, 2006.

[ALL 02] BENJAMIN A. ALLAN, ROBERT C. ARMSTRONG, ALICIA P. WOLFE, JAIDEEP RAY, DAVID E. BERNHOLDT, and JAMES A. KOHL. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14:323–345, 2002.

[ALL 03] GABRIELLE ALLEN, KELLY DAVIS, KONSTANTINOS N. DOLKAS, NIKOLAOS D. DOULAMIS, TOM GOODALE, THILO KIELMANN, ANDRE MERZKY, JAREK NABRZYSKI, JULIUSZ PUKACKI, THOMAS RADKE, MICHAEL RUSSELL, ED SEIDEL, JOHN SHALF, and IAN TAYLOR. Enabling Applications on the Grid - A GridLab Overview. *International Journal on High Performance Computing Applications*, 17(4):449–466, 2003.

[ALL 05] GABRIELLE ALLEN, KELLY DAVIS, TOM GOODALE, ANDREI HUTANU, HARTMUT KAISER, THILO KIELMANN, ANDRE MERZKY, ROB V. VAN NIEUWPOORT, ALEXANDER REINEFELD, FLORIAN SCHINTKE, THORSTEN SCHOTT, ED SEIDEL, and BRYGG ULLMER. "The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid". In *Proceedings of the IEEE*, volume 93, pages 534–550, March 2005.

[ALT 02] MARTIN ALT, HOLGER BISCHOF, and SERGEI GORLATCH. Program Development for Computational Grids Using Skeletons and Performance Prediction. *Parallel Processing Letters*, 12(2):157–174, 2002.

[AND 04] DAVID P. ANDERSON. "BOINC: a system for public-resource computing and storage". In *5th IEEE/ACM International Workshop on Grid Computing*. IEEE Press, November 2004.

[ANT 05] GABRIEL ANTONIU, LUC BOUGÉ, and MATHIEU JAN. JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Computing: Practice and Experience*, 6(3):45–55, September 2005.

[ARM 99] ROB ARMSTRONG, DENNIS GANNON, AL GEIST, KATARZYNA KEAHEY, SCOTT KOHN, LOIS MCINNES, STEVE PARKER, and BRENT SMOLINSKI. "Toward a Common Component Architecture for High-Performance Scientific Computing". In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999. http://www-unix.mcs.anl.gov/ curfman/cca/web/cca_paper.html.

[ATT 05] ISABELLE ATTALI, DENIS CAROMEL, and ARNAUD CONTES. "Deployment-Based Security for Grid Applications". In *The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25*, LNCS. Springer Verlag, 2005.

[BAD 00] B. R. BADRINATH, and PRADEEP SUDAME. "Gathercast: The design and implementation of a programmable aggregation mechanism for the Internet". In *Proceedings of IEEE International Conference on Computer Communications and Networks (ICCCN)*, 2000.

[BAD 02] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. "Efficient, Flexible, and Typed Group Communications in Java". In *joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36. ACM Press, 2002. ISBN 1-58113-559-8.

[BAD 03] ROSA M. BADIA, JESÚS LABARTA, RAÚL SIRVENT, JOSEP M. PÉREZ, JOSÉ M. CELA, and ROGELI GRIMA. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[BAD 05a] LAURENT BADUEL. "*Typed Groups for the Grid*". PhD thesis, University of Nice Sophia-Antipolis, July 2005.

[BAD 05b] LAURENT BADUEL, FRANÇOISE BAUDE, and DENIS CAROMEL. "Object-Oriented SPMD". In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.

[BAD 05c] LAURENT BADUEL, FRANÇOISE BAUDE, NADIA RANALDO, and EUGENIO ZIMEO. "Effective and Efficient Communication in Grid Computing with an Extension of ProActive Groups". In *JPDC, 7th International Worshop on Java for Parallel and Distributed Computing at IPDPS*, Denver, Colorado, USA, April 2005.

[BAK 99] MARK BAKER, BRYAN CARPENTER, GEOFFREY FOX, SUNG HOON KO, and SANG LIM. "mpiJava: An Object-Oriented Java interface to MPI". In *International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP*, San Juan, Puerto Rico, April 1999.

[BAR 05] TOMAS BARROS. "*Formal specification and verification of distributed component systems*". PhD thesis, Université de Nice - Sophia Antipolis, 2005.

[BAU 00] FRANÇOISE BAUDE, DENIS CAROMEL, FABRICE HUET, and JULIEN VAYSSIÈRE. "Communicating Mobile Active Objects in Java". In *Proceedings of HPCN Europe 2000*, volume 1823 de *LNCS*, pages 633–643. Springer, May 2000.

[BAU 02] FRANÇOISE BAUDE, DENIS CAROMEL, LIONEL MESTRE, FABRICE HUET, and JULIEN VAYSSIÈRE. "Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications". In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[BAU 05] FRANÇOISE BAUDE, DENIS CAROMEL, CHRISTIAN DELBÉ, and LUDOVIC HENRIO. "A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability". In *Proceedings of EuroPar2005*, LNCS, pages 644–653, Lisbon, Portugal, August-September 2005. Springer.

[BAU 06] FRANÇOISE BAUDE, DENIS CAROMEL, MARIO LEYTON, and ROMAIN QUILICI. "Grid File Transfer during Deployment, Execution, and Retrieval". In *Proceeedings of On the Move to Meaningful Internet Systems 2006*, Montpellier, France, 2006.

[BER 01] FRANCINE BERMAN, ANDREW CHIEN, KEITH COOPER, JACK DONGARRA, IAN FOSTER, DENNIS GANNON, LENNART JOHNSSON, KEN KENNEDY, CARL KESSELMAN, JOHN MELLOR-CRUMMEY, DAN REED, LINDA TORCZON, and RICH WOLSKI. The GrADS Project: Software Support for High-Level Grid Application Development. *The International Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[BER 04] FELIPE BERTRAND, and RANDALL BRAMLEY. "DCA: A distributed CCA framework based on MPI". In *Proceedings of HIPS 2004, 9th In- ternational Workshop on High-Level Parallel Programming Models and Supportive Environments, Santa Fe, NM*, 2004.

[BER 05] FELIPE BERTRAND, RANDALL BRAMLEY, KOSTADIN B. DAMEVSKI, JAMES A. KOHL, DAVID E. BERNHOLDT, JAY W. LARSON, and ALAN SUSSMAN. "Data Redistribution and Remote Method Invocation in Parallel Component Architectures". In *Proceedings of the 19th International Parallel and Distributed Processing Symposium: IPDPS*, 2005.

[BIO] Bionets Project. http://www.bionets.eu.

[BLO] STEVENS LE BLOND, ANA-MARIA OPRESCU, and CHEN ZHANG. "Early Application Experience with the Grid Application Toolkit (GAT)".

[BLO 05] STEVENS LE BLOND, ANA-MARIA OPRESCU, and CHEN ZHANG. Early Application Experience with the Grid Application Toolkit. Global Grid Forum Information Document, from the Workshop on Grid Applications: from Early Adopters to Mainstream Users, June 2005.

[BOU 06] HINDE LILIA BOUZIANE, CHRISTIAN PÉ REZ, and THIERRY PRIOL. "Modeling and executing Master-Worker applications in component models". In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Rhodes Island, Greece, 25 April 2006.

[BRU 02] ÉRIC BRUNETON, THIERRY COUPAYE, and JEAN-BERNARD STEFANI. "Recursive and Dynamic Software Composition with Sharing". In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, Malaga (Spain), June 10th-14th 2002.

[BRU 04a] ERIC BRUNETON, THIERRY COUPAYE, MATHIEU LECLERCQ, VIVIEN QUÉMA, and JEAN-BERNARD STEFANI. "An Open Component Model and Its Support in Java". In LNCS, teur, *Seventh International Symposium on Component-Based Software Engineering (CBSE-7)*, volume 3054, pages 7–22, May 2004.

[BRU 04b] ERIC BRUNETON, THIERRY COUPAYE, and JEAN-BERNARD STEFANI. The fractal component model specification, 2004.

[BUL 03] J. M. BULL, L. A. SMITH, C. BALL, L. POTTAGE, and R. FREEMAN. Benchmarking Java against C and Fortran for scientific applications. *Concurrency and Computation: Practice and Experience*, 15:417–430, 2003.

[BUY 04] RAJKUMAR BUYYA, and SRIKUMAR VENUGOPAL. “The Gridbus Toolkit for Service Oriented Grid and Utility Computing: An Overview and Status Report”. In *Proceedings of the First IEEE International Workshop on Grid Economics and Business Models (GECON 2004, April 23, 2004, Seoul, Korea)*, pages 19–36. IEEE Press, 2004.

[CAP 05] FRANCK CAPPELLO, FRÉDÉRIC DESPREZ, MICHEL DAYDE, EMMANUEL JEANNOT, YVON JÉGOU, STÉPHANE LANTERI, NOUREDINE MELAB, RAYMOND NAMYST, PASCALE PRIMET, OLIVIER RICHARD, EDDY CARON, JULIEN LEDUC, and GUILLAUME MORNET. “Grid’5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform”. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing, Grid’2005, November 13-14, 2005, Seattle, Washington, USA*, November 2005.

[CAR] DENIS CAROMEL, CHRISTIAN DELBÉ, LUDOVIC HENRIO, and ROMAIN QUILICI. Dispositifs et procé dé s asynchrones et automatiques de transmission de ré sultats entre objets communicants (asynchronous and automatic continuations of results between communicating objects). French patent FR03 13876 - US Patent Pending.

[CAR 93] DENIS CAROMEL. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[CAR 00] BRIAN CARPENTER, VLADIMIR GETOV, GLENN JUDD, ANTHONY SKJELLUM, and GEOFFREY FOX. MPJ: MPI-like Message Passing for Java. *Concurrency: Practice and Experience*, 12:1019–1038, 2000.

[CAR 04] DENIS CAROMEL, LUDOVIC HENRIO, and BERNARD SERPETTE. Asynchronous and Deterministic Objects. pages 123–134, 2004.

[CAR 05] DENIS CAROMEL, and LUDOVIC HENRIO. “*A Theory of Distributed Objects*”. Springer-Verlag, 2005.

[CAR 06] DENIS CAROMEL, ALEXANDRE DI COSTANZO, CHRISTIAN DELBÉ, and MATTHIEU MOREL. “Dynamically-Fulfilled Application Constraints through Technical Services - Towards Flexible Component Deployments”. In *Proceedings of HPC-GECO/CompFrame 2006, HPC Grid programming Environments and COmponents - Component and Framework Technology in High-Performance and Scientific Computing*, Paris, France, June 2006. IEEE.

[CCA] CCA Forum homepage. http://www.cca-forum.org/.

[CHI 02] KENNETH CHIU, MADHUSUDHAN GOVINDARAJU, and RANDALL BRAMLEY. “Investigating the limits of soap performance for scientific computing”. volume 00, page 246, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[CLA 01] MICHAEL CLARKE, GEOFF COULSON, GORDON BLAIR, and NIKOS PARLAVANTZAS. “An Efficient Component Model for the Construction of Adaptive Middleware”. In *Proceedings of Middleware 2001*, 2001.

[COL 89] MURRAY COLE. “*Algorithmic Skeletons: Structured Management of Parallel Computation*”. MIT Press, 1989.

[COP] MASSIMO COPPOLA, MARCO DANELUTTO, SÉBASTIEN LACOUR, CHRISTIAN PÉREZ, THIERRY PRIOL, NICOLA TONELLOTTO, , and CORRADO ZOCCOLO. " Towards a common deployment model for Grid systems ". To appear in Springer CoreGRID series.

[COR] CoreGrid Network of Excellence. http://www.coregrid.net.

[COR 04] IBM CORPORATION. An architectural blueprint for autonomic computing. http://www-128.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html, october 2004.

[DAM 03] KOSTADIN DAMEVSKI. Parallel RMI and M-by-N data redistribution using an IDL compiler. Master's thesis, The University of Utah, 2003.

[DAN 05] MARCO DANELUTTO, MARCO VANNESCHI, CORRADO ZOCCOLO, NICOLA TONELLOTTO, SALVATORE ORLANDO, RANIERI BARAGLIA, TIZIANO FAGNI, DOMENICO LAFORENZA, , and ALESSANDRO PACCOSI. " *Future Generation Grids, CoreGRID series* ", chapter HPC Application Execution on Grids, page 26382. Springer-Verlag, 2005.

[DAV 06] PIERRE-CHARLES DAVID, and THOMAS LEDOUX. " Safe Dynamic Reconfigurations of Fractal Architectures with FScript ". In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[DEN 04] ALEXANDRE DENIS, CHRISTIAN PÉREZ, THIERRY PRIOL, and ANDRÉ RIBES. " Bringing high performance to the CORBA component model ". In *SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

[DES 04] FRÉDÉRIC DESPREZ. " DIET: Building Problem Solving Environments for the Grid ". In *HiPC*, page 4, 2004.

[DUM 06] C. DUMITRESCU, DICK H. J. EPEMA, JAN DUNNWEBER, and SERGEI GORLATCH. " Reusable Cost-based Scheduling of Grid Workflows Operating on Higher-Order Components ". Technical report TR-0044, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, July 2006.

[DUN 04] JAN DUNNWEBER, and SERGEI GORLATCH. " HOC-SA: A Grid Service Architecture for Higher-Order Components ". pages 288–294, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[ECL] Eclipse IDE Project. http://www.eclipse.org.

[EER 03] P. EEROLA, B. KONYA, O. SMIRNOVA, T. EKELOF, M. ELLERT, J. R. HANSEN, J. L.NIELSEN, A. WAANANEN, A. KONSTANTINOV, J. HERRALA, M. TUISKU, T. MYKLEBUST, F. OULD-SAADA, and B. VINTER. " The NorduGrid production Grid infrastructure, status and plans ". In *Proceeding of the 4th International Workshop on Grid Computing (GRID'03)*. IEEE, 2003.

[FAS 02] JEAN-PHILIPPE FASSINO, JEAN-BERNARD STEFANI, JULIA LAWALL, and GILLES MULLER. " Think: A software framework for component-based operating system kernels ". In *Proceedings of the Usenix Annual Technical Conference*, June 2002.

[FEL 96] PASCAL FELBER, BENOÎT GARBINATO, and RACHID GUERRAOUI. " The Design of a CORBA Group Communication Service ". In *Symposium on Reliable Distributed Systems*, pages 150–159. IEEE Computer Society, October 1996.

[FOS 98] IAN FOSTER, and CARL KESSELMAN, teurs. “*The Grid: Blueprint for a New Computing Infrastructure*”. Morgan Kaufmann Publishers, 1998.

[FOS 01] IAN FOSTER, CARL KESSELMAN, and STEVEN TUECKE. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15, 2001.

[FOS 02] IAN FOSTER, CARL KESSELMAN, JEFFREY NICK, and STEVE TUECKE. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. www.globus.org/research/papers/ogsa.pdf., 2002.

[FOS 05a] IAN FOSTER. “Globus Toolkit Version 4: Software for Service-Oriented Systems”. In *IFIP International Conference on Network and Parallel Computing*, number 3779 in LNCS. Springer-Verlag, 2005.

[FOS 05b] IAN FOSTER. Service-Oriented Science. *Science*, 308, 2005.

[FOX 03] GEOFFREY FOX, MARLON PIERCE, DENNIS GANNON, and MARY THOMAS. “Overview of Grid Computing Environments”. Technical report, Global Grid Forum document, 2003. `http://www.ggf.org/documents/final.htm`.

[FRAa] Fractal specification. http://fractal.objectweb.org/specification/index.html.

[FRAb] Fractal tutorial. http://fractal.objectweb.org/tutorial/index.html.

[FRAc] ECOOP’06 Fractal Workshop. http://fractal.objectweb.org/doc/ecoop06/index.html.

[FRA06] Discussion on Fractal ADL from the Fractal mailing list. http://www.objectweb.org/wws/arc/fractal/2006-05/, May 2006.

[FRU 01a] MICHAEL FRUMKIN, and ROB F. VAN DER WIJNGAART. “NAS Grid Benchmarks: A Tool for Grid Space Exploration”. In *Proceedeing of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 ’01)*. IEEE Computer Society, 2001.

[FRU 01b] MICHAEL FRUMKIN, and ROB F. VAN DER WIJNGAART. “NAS Grid Benchmarks: A Tool for Grid Space Exploration”. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC ’01)*, volume 00, page 0315, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[GAN 02] DENNIS GANNON, RANDALL BRAMLEY, GEOFFREY FOX, SHAVA SMALLEN, AL ROSSI, RACHANA ANANTHAKRISHNAN, FELIPE BERTRAND, KEN CHIU, MATT FARRELLEE, MADHU GOVINDARAJU, SRIRAM KRISHNAN, LAVANYA RAMAKRISHNAN, YOGESH SIMMHAN, ALEK SLOMINSKI, YU MA, CAROLINE OLARIU, and NICOLAS REY-CENVAZ. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Cluster Computing*, 5(3), 2002.

[GC] GridComp project. http://gridcomp.ercim.org/.

[GEI 94] AL GEIST, ADAM BEGUELIN, JACK DONGARRA, WEICHENG JIANG, ROBERT MANCHEK, and VAIDYALINGAM S. SUNDERAM. “*PVM: Parallel Virtual Machine*”. MIT Press, 1994.

[GEI 97] G. A. GEIST, JAMES A. KOHL, and PHILIP M. PAPADOPOULOS. CUMULVS: Providing fault tolerance, visualization and steering of parallel applications. *The International Journal of High Performance Computing Applications*, 11(3):224–236, 1997.

[GGF04] Global Grid Forum, Available: http://forge.gridforum.org/projects/saga-rg/. "*Simple API for Grid Applications Research Group*", 2004.

[GOO] TOM GOODALE, SHANTENU JHA, HARTMUT KAISER, THILO KIELMANN, PASCAL KLEIJER, GREGOR VON LASZEWSKI, CRAIG LEE, ANDRE MERZKY, HRABRI RAJIC, and JOHN SHALF. SAGA: A Simple API for Grid Applications, High-Level Application Programming on the Grid. http://www.cs.vu.nl/ kielmann/papers/saga-sc05.pdf. submitted for publication.

[GOR 04] SERGEI GORLATCH. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Transactions on Programming Languages and Systems*, 26(1):47–56, January 2004.

[GPT] 2nd Grid Plugtests. http://www.etsi.org/plugtests/History/2005GRID.htm.

[GPT05] Grid Plugtest: Interoperability on the Grid. Grid Today online, January 2005.

[GRA 04] PAUL GRACE, GEOFF COULSON, GORDON BLAIR, LAURENT MATHY, WAI KIT YEUNG, WEI CAI, DAVID DUCE, and CHRIS COOPER. "GRIDKIT: Pluggable Overlay Networks for Grid Computing". 2004.

[GRO] OBJECT MANAGEMENT GROUP. Corba Component Model Specification. http://www.omg.org/technology/documents/formal/components.htm.

[GRO 03] OBJECT MANAGEMENT GROUP. Data Parallel CORBA Specification. OMG Specification, January 2003.

[HER 00] PETER HERZUM, and OLIVER SIMS. "*Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise*". John Wiley & Sons, Inc., New York, NY, USA, 2000.

[HUA 03] YAN HUANG, IAN TAYLOR, DAVID W. WALKER, and ROBERT DAVIES. "Wrapping Legacy Codes for Grid-Based Applications". In *17th International Parallel and Distributed Processing Symposium: IPDPS*, page 139. IEEE Computer Society, 2003.

[HUE 04] FABRICE HUET, DENIS CAROMEL, and HENRI E. BAL. "A High Performance Java Middleware with a Real Application". In *Proceedings of the Supercomputing conference*, Pittsburgh, Pensylvania, USA, November 2004.

[IBI] Ibis Project. http://www.cs.vu.nl/ibis/index.html.

[ISO 95] ISO/IEC. Open Distributed Processing- Reference Model - Part 1: Overview. International Standard 10746-1 Itu-T Recommendation X.901, 1995.

[JIN 04] HAI JIN. "ChinaGrid: Making Grid Computing a Reality". In *7th International Conference on Asian Digital Libraries, ICADL 2004, Shanghai, China, December 13-17, 2004*, number 3334 in LNCS. Springer, 2004.

[JUG 04] ALEXANDRU JUGRAVU, and THOMAS FAHRINGER. "JavaSymphony, a Programming Model for the Grid". In *International Conference on Computational Science*, pages 18–25, 2004.

[JXT] JXTA Project. http://www.jxta.org/.

[KAR 03] NICHOLAS KARONIS, BRIAN TOONEN, and IAN FOSTER. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, journal =

Journal of Parallel and Distributed Computing (JPDC). 63(5):551–563, May 2003.

[KEA 97] KATARZYNA KEAHEY, and DENNIS GANNON. “PARDIS: A Parallel Approach to CORBA”. In *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1997.

[KEA 01] K. KEAHEY, P. FASEL, and S. MNISZEWSKI. “PAWS: Collective interactions and data transfers”. In *Proceedings of the High Performance Distributed Computing Conference*, 2001.

[KEL 03] RAINER KELLER, BETTINA KRAMMER, MATTHIAS S. MUELLER, MICHAEL M. RESCH, and EDGAR GABRIEL. Towards efficient execution of MPI applications on the Grid: porting and optimization issues. *Journal of Grid Computing*, 1(2):133–149, 2003.

[KIE 99] THILO KIELMANN, RUTGER F. H. HOFMAN, HENRI E. BAL, ASKE PLAAT, and RAOUL A. F. BHOEDJANG. “MagPIe: MPI's collective communication operations for clustered wide area systems”. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, New York, NY, USA, 1999. ACM Press.

[KOH 01] SCOTT KOHN, GARY KUMFERT, JEFF PAINTER, , and CAL RIBBENS. “Divorcing Language Dependencies from a Scientific Software Library”. In *10th SIAM Conference on Parallel Processing*, 2001.

[KRI 04] SRIRAM KRISHNAN, and DENNIS GANNON. “XCAT3: A Framework for CCA Components as OGSA Services”. In *9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, 2004.

[KUM 06] GARY KUMFERT, DAVID E. BERNHOLDT, THOMAS EPPERLY, JAMES KOHL, LOIS CURFMAN MCINNES, STEVEN PARKER, , and JAIDEEP RAY. “How the Common Component Architecture Advances Computational Science”. In *Proceedings of Scientific Discovery through Advanced Computing (SciDAC'06)*, 2006.

[LAC 04a] SÉBASTIEN LACOUR, CHRISTIAN PÉREZ, and THIERRY PRIOL. A Software Architecture for Automatic Deployment of CORBA Components Using Grid Technologies, 2004.

[LAC 04b] SÉBASTIEN LACOUR, CHRISTIAN PÉREZ, and THIERRY PRIOL. “Deploying corba components on a computational grid: General principles and early experiments using the globus toolkit”. In WOLFGANG EMMERICH, and ALEXANDER L. WOLF, teurs, *2nd International Working Conference on Component Deployment (CD 2004)*, LNCS. Springer-Verlag, 2004.

[LAF 02] DOMENICO LAFORENZA. Grid programming: some indications where we are headed. *Parallel Computing*, 28:1733–1752, 2002.

[LAN] SEAN LANDIS, and SILVANO MAFFEIS. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1):1997.

[LAS 01] GREGOR VON LASZEWSKI, IAN FOSTER, JAREK GAWOR, and PETER LANE. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13:643–662, 2001.

[LAS 05] GREGOR VON LASZEWSKI. The Grid-Idea and Its Evolution. *Journal of Information Technology*, 47(6):319–329, 2005.

[LAU 04] PHILIPPE LAUMAY. "*Configuration et déploiement d'intergiciel asynchrone sur système hétérogène à grande échelle*". PhD thesis, Institut National Polytechnique de Grenoble, March 2004. in french.

[LAY 04] OUSSAMA LAYAIDA, SLIM BEN ATALLAH, and DANIEL HAGIMONT. A Framework for Dynamically Configurable and Reconfigurable Network-based Multimedia Adaptations. *Journal of Internet Technology, Special Issue on "Real time media delivery over the Internet"*, October 2004.

[LEC 04] MATTHIEU LECLERCQ, VIVIEN QUÉMA, and JEAN-BERNARD STEFANI. "DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs". In *Proceedings of the 3rd Workshop on Reflective and Adaptive Middleware (RM'2004)*, Toronto, Canada, October 2004.

[LEE 01] C. LEE, S. MATSUOKA, D. TALIA, A. SUSSMAN, N. KARONIS, G. ALLEN, and M. THOMAS. A Grid Programming Primer. Draft 2.4 of the Programming Models Working Group presented at the Global Grid Forum 1, March 2001.

[LEE 03] CRAIG LEE, and DOMENICO TALIA. Grid programming models: Current tools, issues and directions, 2003.

[MAA 02] JASON MAASSEN, THILO KIELMANN, and HENRI E. BAL. "GMI: Flexible and Efficient Group Method Invocation for Parallel Programming". In *Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR'02)*, Washington D.C., USA, March 2002.

[MAL 05] MACIEJ MALAWSKI, DAWID KURZYNIEC, and VAIDY S. SUNDERAM. "MOCCA - Towards a Distributed CCA Framework for Metacomputing". In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*. IEEE Computer Society, 2005.

[MAY 03] ANTHONY MAYER, STEVE MCGOUGH, NATHALIE FURMENTO, WILLIAM LEE, STEVEN NEWHOUSE, and JOHN DARLINGTON. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. UK e-Science All Hands Meeting, p. 627–634, Nottingham, UK, Sep. 2003, 2003.

[MCG] S. MCGOUGH, W. LEE, and J. DARLINGTON. ICENI II Architecture. In UK e-Science All Hands Meeting, Nottingham, UK, sep 2005.

[MEN 05] VLADIMIR MENCL, and TOMAS BURES. "Microcomponent-Based Component Controllers: A Foundation for Component Aspects". In *Proceedings of 12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, pages 729–737. IEEE Computer Society Press, December 2005.

[MIC a] MICROSOFT. COM Framework. http://www.microsoft.com/com/.

[MIC b] MICROSOFT. .Net Framework. http://www.microsoft.com/net/.

[MIC c] SUN MICROSYSTEMS. Enterprise Java Beans. http://java.sun.com/products/ejb/.

[MIC d] SUN MICROSYSTEMS. Java Community Process. http://jcp.org/en/home/index.

[MIN 97] SAVA MINTCHEV, and VLADIMIR GETOV. "Towards Portable Message Passing in Java: Binding MPI". In *Proceedings of the 4th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 135–142, London, UK, 1997. Springer-Verlag.

[MPI94] "MPI: A Message-Passing Interface Standard". Technical report, MPI Forum, University of Tennessee, Knoxville, Tennessee, June 1994.

[NAR] "*The Naregi project*". http://www.naregi.org/index_e.html.

[NEL 01] ARNOLD NELISSE, THILO KIELMANN, HENRI E. BAL, and JASON MAASSEN. "Object-based Collective Communication in Java". In *Joint ACM Java Grande - ISCOPE Conference*, pages 11–20, Palo Alto, California, USA, June 2001. ACM Press. ISBN 1-58113-359-6.

[NIE 02] ROB V. VAN NIEUWPOORT, JASON MAASSEN, RUTGER HOFMAN, THILO KIELMANN, and HENRI E. BAL. "Ibis: an efficient Java-based grid programming environment". In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 18–27, Seattle, Washington, USA, November 2002.

[NIE 05] ROB V. VAN NIEUWPOORT, JASON MAASSEN, THILO KIELMANN, and HENRI E. BAL. Satin: Simple and Efficient Java-based Grid Programming. *Scalable Computing: Practice and Experience*, 6(3):19–32, September 2005.

[PAR 93] CRAIG PARTRIDGE, TREVOR MENDEZ, and WALTER MILLIKEN. Host anycasting service. RFC 1546, 1993.

[PAR 98] S.G. PARKER, and C.R. JOHNSON. "An integrated problem solving environment: The SCIRun computational steering system". In *31st Hawaii International Conference on System Sciences*, volume 7, pages 147–156, 1998.

[PAR 05] MANISH PARASHAR, and JAMES .C. BROWNE. "Conceptual and Implementation Models for the Grid". In *Proceedings of the IEEE, Special Issue on Grid Computing*, volume 93, page 653 668. IEEE Press, March 2005.

[PÉR 03] CHRISTIAN PÉREZ, THIERRY PRIOL, and ANDRÉ RIBES. A Parallel CORBA Component Model for Numerical Code Coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417–429, 2003.

[PÉR 04] CHRISTIAN PÉREZ, THIERRY PRIOL, and ANDRÉ RIBES. "PACO++: A Parallel Object Model for High Performance Distributed Systems". In *Proceedings of the 37th Hawaii International Conference on System Sciences (HICCS'04)*, 2004.

[PLA 98] FRANTISEK PLASIL, DUSAN BALEK, and RADOVAN JANECEK. "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating". In IEEE CS PRESS, teur, *Proceedings of ICCDS'98*, 1998. Annapolis, Maryland, USA.

[PRI 98] THIERRY PRIOL, and CHRISTOPHE RENÉ. "Cobra: A CORBA-compliant Programming Environment for High-Performance Computing". In LNCS, teur, *Proceedings of Euro-Par'98 Conference*, number 1470, pages 1114–1122. Springer-Verlag, September 1998.

[PRO] "*ProActive web site*". http://proactive.objectweb.org.

[RAM 98] RAJESH RAMAN, MIRON LIVNY, and MARVIN H. SOLOMON. "Matchmaking: Distributed Resource Management for High Throughput Computing". In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 140–, July 1998.

[REN 99] CHRISTOPHE RENÉ, and THIERRY PRIOL. "MPI Code Encapsulation using Parallel CORBA Object". In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 3–10, 1999.

[RIB 04] ANDRÉ RIBES. "*Contribution à la conception d'un modèle de programmation parallèle et distribué et sa mise en oeuvre au sein de plates-formes orientées objet et composant*". PhD thesis, IRISA, Université de Rennes 1, 2004. in french.

[ROU 06] ROMAIN ROUVOY, and NICOLAS PESSEMIER. Fraclet Annotation Framework Documentation. http://fractal.objectweb.org/tutorials/fraclet/index.html, June 2006.

[SCA] Service Component Architecture Specification. http://www-128.ibm.com/developerworks/library/specification/ws-sca/.

[SEI 06] LIONEL SEINTURIER, NICOLAS PESSEMIER, LAURENCE DUCHIEN, and THIERRY COUPAYE. "A Component Model Engineered with Components and Aspects". In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, 2006.

[SEK 96] S. SEKIGUCHI, M. SATO, H. NAKADA, S. MATSUOKA, and U. NAGASHIMA. "Ninf: Network-based information library for globally high performance computing". In *Parallel Object-Oriented Methods and Applications (POOMA)*, pages 39–48, 1996. `http://ninf.etl.go.jp`.

[SEY 02] KEITH SEYMOUR, HIDEMOTO NAKADA, SATOSHI MATSUOKA, JACK DONGARRA, CRAIG A. LEE, and HENRI CASANOVA. "Overview of GridRPC: A Remote Procedure Call API for Grid Computing". In *GRID*, pages 274–278, 2002.

[SEY 05] KEITH SEYMOUR, ASIM YARKHAN, SUDESH AGRAWAL, and JACK DONGARRA. "NetSolve: Grid Enabling Scientific Computing Environments". In L. GRANDINETTI, teur, *Grid Computing and New Frontiers of High Performance Processing*. Elsevier, 2005.

[SHA 93] MARY SHAW. "Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status". In SPRINGER-VERLAG 1994 LECTURE NOTES IN COMPUTER SCIENCE, teur, *Proceedings of Workshop on Studies of Software Design*, 1993.

[SLO 06] ALEKSANDER SLOMINSKI. On using BPEL extensibility to implement OGSI and WSRF Grid workflows. *Concurrency and Computation: Practice & Experience*, 18(10):1229 – 1241, August 2006.

[SNA 03] ALLAN SNAVELY, GREG CHUN, HENRI CASANOVA, ROB F. VAN DER WIJNGAART, and MICHAEL A. FRUMKIN. Benchmarks for grid computing: a review of ongoing efforts and future directions. *SIGMETRICS Perform. Eval. Rev.*, 30(4):27–32, 2003.

[SQU 96] JEFFREY M. SQUYRES, BRIAN C. MCCANDLESS, and ANDREW LUMSDAINE. “Object Oriented MPI: A Class Library for the Message Passing Interface”. In *Parallel Object-Oriented Methods and Applications (POOMA '96)*, Santa Fe, 1996.

[SRI 05] LATHA SRINIVASAN, and JEM TREADWELL. An Overview of Service-oriented Architecture, Web Services and Grid Computing. Hewlett-Packard White Paper, 2005.

[SUN 02] VAIDY SUNDERAM, and DAWID KURZYNIEC. “Lightweight self-organizing frameworks for metacomputing”. In *Proceedings of the 11th International Symposium on High Performance Distributed Computing (HPDC)*. IEEE Computer Society, 2002.

[SZY 96] CLEMENS SZYPERSKI, and UNO. PFISTER. WCOP'96 Workshop Report. Worskshop Reader ECOOP'96, p127-130, june 1996.

[SZY 02] CLEMENT SZYPERSKI. “*Component Software : Beyond Object-Oriented Programming*”. Addison-Wesely, 2002.

[TAN 03] YOSHIO TANAKA, HIDEMOTO NAKADA, SATOSHI SEKIGUCHI, TOYOTARO SUZUMURA, and SATOSHI MATSUOKA. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1:41–51, 2003.

[TAY 05] IAN TAYLOR, MATTHEW SHIELDS, IAN WANG, and ANDREW HARRISON. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[TER] TeraGrid project. http://www.teragrid.org.

[THA 05] DOUGLAS THAIN, TODD TANNENBAUM, and MIRON LIVNY. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[UNI] Unicore Forum. http://www.unicore.org.

[VOG06] A conversation with Werner Vogels: Learning from the Amazon technology platform. ACM Queue, vol 4, no 4, May 2006.

[WAL 94] JIM WALDO, GEOFF WYANT, ANN WOLLRATH, and SAM KENDALL. “A Note on Distributed Computing”. Technical report TR-94-29, Sun Microsystems Laboratories, Inc., 1994.

[WSR04] WS-Resource Framework. http://www.globus.org/wsrf/, 2004.

[YOU 03] LAURIE YOUNG, STEPHEN MCGOUGH, STEVEN NEWHOUSE, and JOHN DARLINGTON. “Scheduling Architecture and Algorithms within the ICENI Grid Middleware”. In *UK e-Science All Hands Meeting*, pages 5–12, September 2003.

[YU 05] JIA YU, and RAJKUMAR BUYYA. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record*, 34(3):44–49, 2005.

[ZHA 04] KEMING ZHANG, KOSTADIN DAMEVSKI, VENKATANAND VENKATACHALAPATHY, and STEVEN G. PARKER. “SCIRun2: A CCA Framework for High Performance Computing”. In *Proceedings of the Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'04)*, 2004.

[ZHA 06] LI ZHANG, and MANISH PARASHAR. Seine: A Dynamic Geometry-based Shared Space Interaction Framework for Parallel Scientific Applications. *Concurrency and Computations: Practice and Experience*, 2006.

Components for Grid Computing

Abstract

This thesis aims at facilitating the design and deployment of distributed applications on Grids, using a component-based programming approach.

The specific issues in Grid computing addressed by the proposal of this thesis are: complexity of design, deployment, flexibility and high performance. We propose and justify a component model and an implementation framework. Our component model grounds on the Fractal component model and the active object model, It takes advantage of, first, the hierarchical model, well defined semantics and extensibility of the Fractal model, and second, the identification of components as configurable activities. We define a deployment model based on the concept of virtual architectures, and we propose primitives for collective communications through the specification of collective interfaces. Collective interfaces handle data distribution, parallelism and synchronization of invocations. They establish a basis for defining complex interactions between multiple components.

We realized an implementation of this model on top of the ProActive Grid middleware, therefore benefiting from underlying features of ProActive. We demonstrate the scalability and efficiency of the framework by developing and deploying on several hundred nodes a compute and communication-intensive application, and we take advantage of the collective interfaces to develop a component-based SPMD application with benchmarks.

Keywords : Component-Based Programming, Grid Computing, Collective Communications

---

Composants pour la Grille

Résumé

L’objectif de cette thèse est de faciliter la conception et le déploiement d’applications distribuées sur la Grille, en utilisant une approche orientée composants.

Les problématiques du calcul sur grilles abordées dans notre proposition sont: la complexité de conception, le déploiement, la flexibilité et la performance. Nous proposons et justifions un modèle de composants et son implantation. Le modèle proposé repose sur le modèle de composants Fractal et sur le modèle des objets actifs. Il bénéficie d’une part, de la structure hiérarchique et de la définition précise du modèle Fractal, et d’autre part, de l’identification des composants comme activités configurables. Nous proposons un modèle de déploiement et nous spécifions un ensemble de primitives pour les communications collectives, grâce à la définition d’interfaces collectives. Les interfaces collectives permettent de gérer la distribution des données, le parallélisme et la synchronisation des invocations.

Nous avons développé une implantation du modèle proposé avec l’intergiciel de grille ProActive. Le framework de composants bénéficie ainsi des fonctionnalités sous-jacentes offertes par l’intergiciel ProActive. Nous démontrons la capacité de passage à l’échelle et l’efficacité de notre framework en déployant sur plusieurs centaines de machines des applications intensives en termes de calcul et de communications. Nous mettons à profit les interfaces collectives pour développer une application SPMD à base de composants, dont nous évaluons les performances.

Mots-clefs : Programmation par Composants, Grilles de Calcul, Communications Collectives