Annals of Telecommunications manuscript No.
(will be inserted by the editor)

GCM: A Grid Extension to Fractal for Autonomous Distributed Components

Françoise Baude · Denis Caromel · Cédric Dalmasso · Marco Danelutto · Vladimir Getov · Ludovic Henrio · Christian Pérez

Abstract This article presents an extension of the Fractal component model targeted at programming applications to be run on computing grids: the Grid Component Model (GCM). First, to address the problem of deployment of components on the Grid, deployment strategies have been defined. Then, as Grid applications often result from the composition of a lot of parallel (sometimes identical) components, composition mechanisms to support collective communications on a set of components are introduced. Finally, because of the constantly evolving environment and requirements for Grid applications, the GCM defines a set of features intended to support component autonomicity. All these aspects are developed in this paper with the challenging objective to ease the programming of Grid applications, while allowing GCM components to also be the unit of deployment and management.

I Introduction

Grid computing raise a lot of challenges for programming models, because it consists in programming and running applications running over *large scale heterogeneous* resources that evolve *dynamically*. The GCM (Grid Component model) addresses the characteristic challenges in terms of programmability, interoperability, code reuse and efficiency. Programming large-scale distributed systems as Grids can be considered as a matter of distributed services deployment and further integration. In this paper,

F. Baude, D. Caromel, C. Dalmasso and L. Henrio
INRIA Sophia-Antipolis, I3S, Univ. de Nice Sophia-Antipolis, CNRS, Sophia Antipolis, France
E-mail: first.last@inria.fr

M. Danelutto
University of Pisa, Pisa, Italy
E-mail: marcod@di.unipi.it

V. Getov
Harrow School of Computer Science, University of Westminster, London, U.K.
E-mail: V.S.Getov@westminster.ac.uk

C. Pérez
INRIA/IRISA, Rennes, France
E-mail: christian.perez@inria.fr

we advocate the idea that a hierarchical and distributed software component based approach is an effective solution to this.

I.1 Objectives

The research challenges dealt with by the GCM are thus the support at the application level for *heterogeneity*, *large-scale distribution* and *dynamic management and adaptivity* by means of a component model to provide a high *programmability of Grid applications*.

Programmability deals with the expressive power of the language mechanisms that are offered to the programmers, and what is the burden for them to effectively use those mechanisms. A short overview of current proposed grid frameworks makes us believe that it is in the programmability dimension that resides the greatest divergence between those solutions. Schematically, these solutions range

- from low-level message-passing (for example MPI), RPC or RMI based traditional parallel and distributed programming models – simply ported to tackle grid issues – by which the program itself dictates and orchestrates the parallelism and distribution of computing and communicating entities [24];
- to solutions in which the orchestration or choreography of the set of parallel and distributed entities is guided from the extern of these entities, not necessarily in a centralised manner by using for example workflow languages and programming [23].

We think that these two categories are not exclusive, because the spectrum of applications that could benefit from running on grids is not closed. The purpose of the GCM is to reconcile those two extreme points of view: a component approach allows both explicit communications between distributed entities like in MPI, and high-level management of the distribution of those entities and their interactions, like in workflows. GCM mainly focuses on the programmability of end-user Grid applications, but is also suited to program tools and middleware in the computing Grid context: those can be designed and implemented as GCM components featuring specific services.

So, we aim at proposing a solid and adequate parallel and distributed programming model laying the foundation for building any form of grid application. Its qualities must be those of *expressiveness*, *extensibility*, solid *theoretical foundation*, and suitability for *optimisation* and competitive implementations. In light of this, we selected the Fractal component model as the starting point for offering a versatile yet structured and tractable Grid programming model.

I.2 Approach and Contribution

A general issue when designing a component model is the advised granularity of the components: “what is the size of a component?”. This issue is often overridden in the presentation of a component model, but is crucial to understand the decisions taken in such a design. In the case of a hierarchical component model like Fractal, this question becomes “what is the size of a primitive component?”, or “what is the unit of composition?”. Fractal does not impose any granularity for the components, but the concept of binding component [10] and some of the features of the model suggest a fine grained implementation: a primitive component is assimilated to one or a few objects.

The granularity of the component model is to our mind a crucial aspect because it influences the expressive power and the overhead of the component architecture: a fine-grain system increases the ability to compose components, but generally entails additional cost to manage a larger number of entities, and to make them interact.

When addressing distribution aspects of a component model, the same question arises again, but becomes more complex: "what is the relation between the unit of composition (the primitive component) and the unit of distribution?". Like Fractal, the GCM does not enforce precisely any granularity of the components. However, in order to allow GCM primitive components to be also the unit of distribution for a GCM implementation, we consider that GCM component implementations would probably have a coarser granularity than Fractal ones. This difference in the advocated component granularity partially explains why some of the highly used features in a Grid setting as collective communication mechanisms have been defined as first class citizens in the GCM. For example, multicast communication could be expressed in Fractal by relying on binding components but such components would be too small to be used as the unit of distribution. In brief, in GCM *each component is subject to distribution.*

Compared to other component models, the GCM has been conceived around a component granularity that is somehow in middle between small grain Fractal components and very coarse grain ones, like suggested by CCM where a component is of a size comparable to a full-fledged application. Somehow, GCM has been conceived thinking of components of the size of a process (i.e., one or a few threads per primitive component), though it can be used in a much finer or coarser grain way.

To address the challenges expressed above, the GCM is a specification taking the following approach. *Distribution* concerns are specified at the composition level by specific entries in the ADL (Architecture Description Language) relying either on a controlled or on an automatic mapping between computing resources of the infrastructure and primitive components. Many-to-one and one-to-many communications are key mechanisms for optimising communications in a *large-scale* environment; they are also a key *programming constructs* for distributing computations and synchronising their results. This paper also studies the effective combination of one-to-many and many-to-one interfaces: the MxN problem. Finally, *heterogeneous dynamic* large infrastructures require the adaptation of the application and its management to be totally distributed, and consequently, preferably autonomous. For this, GCM extends Fractal with controllers as components, and with the definition of interfaces for autonomicity, to enable the autonomous control to be designed as a component-based system.

The point of view we adopt here is close to Fractal: we are not tied to any programming language, however like in Fractal, we reuse the terminology of object-oriented programming. Components are thought as autonomous service entities exchanging messages or requests according to precisely defined ports (named interfaces in Fractal).

I.3 Foundations

The GCM has been defined by the CoreGRID European Network of Excellence gathering researchers in the area of Grid and Peer-to-Peer technologies. It relies on the following aspects inherited from existing works:

- *Fractal as the basis for the component architecture:* We summarise the characteristics we benefit from Fractal in Section II.1.

– Communication Semantics: GCM components should allow for any kind of communication semantics (e.g., streaming, file transfer, event-based) either synchronous or asynchronous. Of course, for dealing with high latency, asynchronous communications will probably be preferred by most GCM frameworks.

I.4 Outline

This paper starts with an overview of existing component models that can be used in the area of Grid computing, in Section II. Among the central features of the GCM, this article will focus on the most innovative ones:

– Support for deployment: distributed components need to be deployed over various heterogeneous systems. The GCM defines deployment primitives for this. Deployment aspects will be developed in Section III.
– Support for one-to-many, many-to-one, and many-to-many communications: often, Grid applications consist of a lot of similar components that can be addressed as a group, and that can communicate together in a very structured way. The GCM also intends at providing high level primitives for a better design and implementation of such *collective communications* which will be detailed in Section IV.
– Support for non-functional adaptivity and autonomic computation: the Grid is an highly evolving environment, and Grid applications must be able to adapt to those changing runtime conditions. For this reason, we propose to allow for both reconfiguration of the component control aspects, and autonomic computation support. Adaptivity and autonomicity in the GCM will be presented in Section V.

II Other Distributed Component Models

This section reviews the main component models, it first briefly presents what peculiar and interesting features the Fractal abstract component model provides, consequently argumenting why we selected it as the basis for the GCM. Next, we review some other software component models that are targeted at the programming of distributed applications, or even of middleware, taking into account constraints raised by distribution.

II.1 Fractal

Fractal [10] is a general component model which is intended to implement, deploy and manage (i.e. monitor, control and dynamically configure) complex software systems, including in particular operating systems and middleware. Among Fractal's peculiar features, below are those that motivated us to select it as the basis for the GCM.

– Hierarchy (composite components can contain sub-components), to have a uniform view of applications at various levels of abstraction.
– Introspection capabilities, to monitor and control the execution of a running system.
– Reconfiguration capabilities, to dynamically configure a system.

To allow programmers to tune the control of reflective features of components to the requirements of their applications, Fractal is defined as an extensible system.

Fractal comes with a formal specification. It can be instantiated in different languages such as Java and C. In addition, the Fractal specification is a multi-level specification, where depending on the level some of the specified features are optional. That means that the model allows for a continuum of reflective features or *levels of control*, ranging from no control (black-boxes, standard objects) to full-fledged introspection and intercession capabilities (including e.g. access and manipulation of component contents, control over components life-cycle and behaviour, etc).

Fractal already has several implementations in different languages. The GCM is not tied to Fractal's reference implementation (Julia), which is not targeted at distributed architectures. Dream is a library built using Julia Fractal components targeting distribution, but specifically aimed at building message-oriented middleware, and not grid applications or even grid middleware as we intend to do.

To sum up, it is because of its extensible and hierarchical nature that Fractal has been chosen as the basis for the definition of the GCM. Fractal does not constrain the way(s) the GCM can be implemented, but it provides a basis for its formal specification, allowing us to focus only on the Grid-specific features. Eventually, platforms implementing the GCM should constitute suitable grid programming and execution environments. ProActive offers one such implementation [5].

II.2 Distribution-aware component models

This section focuses on some of the main distributed component models and on what is missing in these models in order to fully support a structured approach to Grid programming, underlying the necessity for an innovative and new component model.

Let us first focus on two commonly known models for a component-oriented approach [38] to distributed computing: the *Common Component Architecture (CCA)* [3, 12] and the *CORBA Component Model (CCM)* [33].

- *CCA* has been defined by a group of researchers from laboratories and academic institutions committed to specifying standard component architectures for high performance computing. The basic definition in CCA states that a component "is a software object, meant to interact with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behaviour common to all components within an architecture." Currently the CCA Forum maintains a web-site gathering documents, projects and other CCA-related work (www.cca-forum.org) including the definition of a CCA-specific format of component interfaces (Babel/SIDL – SRPC Interface Description Language) and framework implementations (Ccaffeine, Xcat)
- *CCM* is a component model defined by the Object Management Group (OMG), an open membership for-profit consortium that produces and maintains computer industry specifications such as CORBA, UML, and XMI. The CCM specifications include a Component Implementation Definition Language (CIDL); the semantics of the CORBA Component Model (CCM); a Component Implementation Framework (CIF), which defines the programming model for constructing component implementations and a container programming model. Important work has been performed to turn the CCM in a Grid component model, like GridCCM [18].

In recent years, the US-based CCA initiative brought together a number of efforts in component-related research projects, with the aim of developing an interoperable

Grid component model and extensions for parallelism and distribution [9]. However, the CCA model is non-hierarchical, thereby making it difficult to handle the distributed and possibly large set of components forming a Grid application [22] in a structured way. Indeed, hierarchical organisation of a compound application can prove very useful in getting scalable solutions for management operations pertaining to monitoring, life-cycle, reconfiguration, physical mapping on grid resources, load-balancing, etc. Unfortunately, the CCA model is rather poor with regards to managing components at runtime. It means a CCA component per se does not have to expose standard interfaces dedicated to non-functional aspects as it is the case for Fractal, and consequently GCM components. This makes it hard to realize certain features, for instance, dynamic reconfiguration based on observed performance or failures. However, some implementations of the model, like e.g. XCAT, can provide some additional components (like an Application Manager) dedicated to manage the non-functional aspects of a CCA-based application. But this has to be considered as additional and optional features, not defined by the component model, so preventing interoperability between CCA components running onto different platforms. Consequently, we think that the GCM is a richer programming model than CCA, and allow the effective design and management of distributed applications at a grid scale.

CCM presents the same limitations than CCA with the exception that CCM handles quite well the heterogeneity of resources. In CCM, the ADL is able to deal with distributed resources but it is outside the scope of the specifications to describe how such a description has been generated. However, this task requires a high level of knowledge of the application structure as well as the resource properties. This approach is not satisfactory for Grids where resources are provided dynamically. Hence, while CCM has some very interesting features for Grids – in particular because CCM has been designed for distributed applications – it appears as a model where distribution is too much coupled to the resources for Grid applications.

Even if CCA and CCM components can fit into a distributed infrastructure, they are not designed as being per se distributed, and possibly parallel entities to be mapped onto a set of grid resources, nor having the capability to self-adapt to the changing context. By contrast, the Enterprise Grid Alliance effort [40] is an attempt to derive a common model adopting Grid technologies for enhancing the enterprise and business applications. The model, which is aligned with industry-strength requirements, strongly relies on component technology along with necessary associations with component-specific attributes, dependencies, constraints, service-level agreements, service-level objectives and configuration information. One of the key features that the EGA reference model suggests is the life-cycle management of components which could be governed by policies and other management aspects. The level of this specification, however, is very coarse-grain focusing on system integration support rather than providing an abstract model and specification for Grid programming which is the main goal of GCM.

Most of Grid oriented component models use components to wrap complete, possibly parallel, applications. This is sufficient to build new grid-wide HPC applications, e.g. multi-disciplinary ones, by composition of a few separate software modules. This also means that a such components must not be considered as the unit of distribution, but as a coarse-grain unit wrapping a full-fledged software exposed as a Grid service, to be composed with a few others. On the contrary, a GCM primitive component is a well delimited unit of distribution and management at the scale of the grid, and a GCM composite component is a suitable abstraction to hierarchically handle at once any sort of distributed and parallel composition, including ones that may be formed of

a very large set of software units spread all over the grid and running in parallel. Of course, this does not prevent a primitive GCM component to itself wrap a legacy, e.g. MPI, parallel application, but in this case it is clear that the resulting set of parallel processes, probably co-located on a same cluster of machines, is under the management responsibility of the primitive component itself.

In terms of Grid middleware, there have been a few platforms such as ICENI [21] that enable users to build grid applications out of software components. On several platforms, applications running on the grid are interconnected by some kind of collective binding mechanisms, notably in Xcat and ICENI. However, most of the component-oriented platforms that we are aware of, support components at application level only without any componentisation at runtime environment level. Instead, the design of ICENI follows the classical service oriented architecture (SOA) approach [20]. Obviously, a side-effect of such SOA-based approaches is the strong importance given to interoperability through, for example, the WSDL-based exportation of the component interfaces. Interoperability is also recognised as a key aspect of the GCM, in order to be capable to loosely connect external applications based upon any kind of technology to a GCM-based one [19].

One of the exceptions among the existing component-oriented platforms is the GRIDKIT project [15]. In GRIDKIT the middleware itself is designed as components, derived from OpenCOM. In addition, the GRIDKIT team identified the need for support of multiple complex communication paradigms, non-functional (horizontal) services, autonomicity, and reconfiguration. The GCM addresses these concerns but at a different level by providing corresponding support as integral part of the component model itself so that GCM-based Grid middleware and applications can benefit from those features. Thus, an interesting perspective could be to adopt the GCM in future versions of the GRIDKIT middleware in order to benefit from these advanced features both at the component model level and at the middleware one. GCM has already proved to be efficient for conceiving a Grid runtime support inside the CoreGRID project [13].

Compared to related works, GCM originality lies in its adopted model, at the level of components themselves, for deployment, collective communications, adaptivity and autonomicity.

III Deploying Components

GCM applications are primarily designed to be run on Grids, that is to say on a complex and dynamic distributed systems. Hence, a major question is how to express the mapping of the components on the resources. Grid environments usually provide job schedulers whose task is to compute when and where to launch an application. However, job schedulers are system level entities: as such they are only able to deal with simple jobs such as sequential jobs and MPI-like jobs for the most advanced. It is far behind the current state of the art of the schedulers to deal with complex structures such as a hierarchy of distributed components. Hopefully, Grid environments also provide information services. Hence, it is possible to imagine a dedicated deployment service that can take care of selecting adequate resources for an application.

Component models usually enable the description of the initial structure of an application thanks to some ADL. However, for distributed platforms, ADL files include the name of the resources. It is well suited for a *particular* deployment of an application on a known set of resources. But, it is inappropriate to have to change these files

each time the application is deployed on a different platform, whereas the application architecture and implementation did not change. Therefore, explicitly mention the name of resources inside an ADL is not well suited to describe a Grid application.

The GCM provides two strategies, a simple and a more advanced one, to deal with this issue. The first strategy is based on the virtual node concept. It aims at enabling a logical grouping of the components on a virtual infrastructure. The second strategy aims at not presenting any infrastructure concept to the application. The remaining of this section presents them.

III.1 Controlled Mapping through Virtual Nodes

A first strategy for supporting deployment is to rely on virtual nodes. Virtual nodes are abstractions allowing a clear separation between design infrastructure and physical infrastructure. This concept already exists both in the standard Fractal ADL and the ProActive middleware. Virtual nodes can be used in the ADL and they can abstract away names, but also creation and connection protocols. Consequently applications remain independent from connection protocols and physical infrastructure. A virtual node contains one or more nodes. A node represents a location where a component can be created and executed. This can be a single physical machine (a host), or, in the case of a multi-processor / multi-core machine, a single processor or a single core within a machine.

The *virtual-node* element, in ADL files, offers distributed deployment information. To better specify the deployment constraints on a component, the standard Fractal ADL has been extended. The *cardinality* attribute has been added to the *virtual-node* element. And, in addition to this element, the GCM add the possibility to export and compose virtual nodes in the *exportedVirtualNodes* element. We will describe how these elements can be used to control the component/virtual-node mapping in ADL files.

The syntax is similar to the Fractal ADL, features specific to the GCM are:

- Virtual nodes have a cardinality: either *single* or *multiple*. *Single* means the virtual node in the deployment descriptor should contain one node; *Multiple* means the virtual node in the deployment descriptor should contain more than one node. For example, the following element in a component definition indicates that we want create the component in the virtual node *client-node* which contains one node.

```
<virtual-node name="client-node" cardinality="single"/>
```

- Virtual nodes can be exported and composed. Export and compose allow respectively to rename and merge virtual nodes. This extends re-usability of existing components. When exported, a virtual node can take part in the composition of other exported virtual nodes. The following composition code creates a new virtual node named *client-node*, composed from two virtual nodes, *client1* and *client2*, defined in components *c1* and *c2*.

```
<exportedVirtualNodes>
<exportedVirtualNode name="client-node">
<composedFrom>
 <composingVirtualNode component="c1" name="client1"/>
 <composingVirtualNode component="c2" name="client2"/>
</composedFrom>
</exportedVirtualNode>
</exportedVirtualNodes>
```

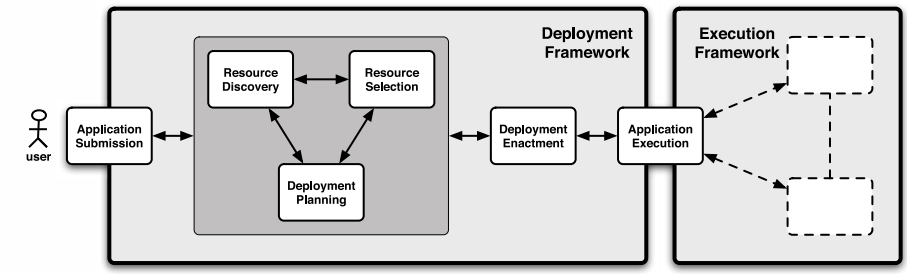


Fig. 1 Deployment process for automatic mapping

Then, mapping from virtual nodes to the infrastructure is defined in separate files, called deployment descriptors. Those files describe the real infrastructure and the way to acquire resources; we do not detail the format of deployment descriptors here, see [5]. Components are deployed on a node included in the virtual node that is specified in their definition; it has to appear in the deployment descriptor unless this virtual node is exported.

A component will be instantiated on the node associated to the virtual node given in its ADL (modulo the renaming entailed by exportation). In case several components use the same virtual node with a multiple cardinality we don't specify on which node we create each component.

III.2 Automatic Mapping to the Infrastructure

Deployment descriptors provide a mean for expert programmers/deployers to control how a particular application is deployed on a set of resources. Another abstraction step is needed to further decouple an application from the resources. The underlying idea is to let a programmer specify its component assembly within a model without any resource concept, i.e. without any knowledge on the physical architecture. Then, an automatic deployment process is needed to derive a mapping of the components to the available resources. This section reviews the needed steps to achieve such an automatic mapping. It shows that most steps are already provided by current Grid environments and details what is still needed.

Overview. Starting from a description of an application and a user objective function, the deployment process is responsible for automatically performing all the steps needed to start the execution of the application on a set of selected resources. These steps are illustrated in Figure 1. The logical order of the activities is fixed (Submission, Discovery, Planning, Enactment, Execution). Some steps have to be re-executed when the application configuration is changed at run-time. Moreover, the steps in the gray box, that interact closely, can be iterated until a suitable set of resources is found.

The following describes the activities involved in the deployment of an application. This process only takes as input a file describing the components of the application, their interactions, and the characteristics of the required resource.

Application Description. The application may be described in a variant of Fractal ADL, which contains several kinds of data: the description of the component types and their implementations as well as information to guide the mapping of the application onto resources. It may consist of the **resource constraints** – characteristics that resources (computational, storage, network) must possess to execute the application, the **execution platform constraints** – software (libraries, middleware systems) that must be installed to satisfy application dependencies, the **placement policies** – restrictions or hints for the placement of subsets of application processes (e.g. co-location, location within a specific network domain, or network performance requirements), and the **resource ranking** – an objective function provided by the user, stating the optimisation goal of application mapping. Resource ranking is exploited to select the best resource, or set of them, among those satisfying the given requirements for a single application process. Resource constraints can be expressed as *unitary requirements*, that must be respected by a single module or resource (e.g. CPU rata), and as *aggregate requirements*, that a set of resources or a module group must respect at the same time (e.g. all the resources on the same LAN, access to a shared file system); some placement policies are implicitly aggregate requirements. As of today, there is no standard format for describing the constraints, the placement policies, or the resource ranking.

Resource Discovery. This activity finds the resources compatible with the execution of the application. Resources satisfying unitary requirements can be discovered, interacting with Grid Information Services [16]. Then, the information needed to perform resource selection (that considers also aggregate requirements), must be collected, for each suitable resource found. Existing grid technology are quite satisfactory with respect to this point but co-allocation support in grid scheduler is still quite uncommon.

Deployment Planning When information about available resources is collected, the proper resources that will host the execution of the application must be selected, and the different parts of each component have to be mapped on some of the selected resources. This activity also implies satisfying all the aggregate requirements within the application. Thus, repeated interaction with the resource discovery mechanisms may be needed to find the best set of resources, also exploiting dynamic information.

At this point, the user objective function must be evaluated against the characteristics and available services of the resources (expressed in the resource description schema). When appropriate, a resource ranking is established to find a suitable solution.

An abstract deployment plan is computed by gathering the deployment schema of all application components. The abstract plan is then mapped onto the resources, and turned into a concrete plan, identifying all the services and protocols that will be exploited in the next phase on each resource, in order to set up and start the runtime environment of the application.

This step is probably the most challenging one as it requires advanced algorithms (heuristics) to compute a plan as the problem is generally NP-hard.

Deployment Enactment The concrete deployment plan developed in the previous phase is submitted to the execution framework, which is in charge of the execution of the tasks needed to deploy the application. This service must ensure a correct execution of the deployment tasks while respecting the precedences described in the deployment plan. At the end of this phase, the execution environment of the application is ready to start its actual execution. This step is nowadays quite well mastered.

Application Execution The deployment process for adaptive Grid applications does not finish when the application is started. Several activities have to be performed while the application is active. The whole application life-cycle must be managed, in order to support new resource requests for application adaptation, to schedule a restart if a failure is detected, and to release resources when the normal termination is reached. These monitoring and controlling activities are mediated by the autonomic part of the components, which performs some dynamic deployment action.

III.3 Discussion

This section has presented two deployment strategies for a Grid application: one strongly driven by the user, and a much more automatic one. The first deployment strategy provides a mechanism to capture some topological constraints of the mapping of the component hierarchy to the resources. The application can map its elements to the virtual nodes independently of the real resource names: the application is portable. Moreover, the mapping of the virtual nodes to the physical nodes appears at the level of current grid schedulers.

The second deployment strategy aims at providing an automatic mapping of the application on the resources. It requires to extend ADL with constraints and placement policies as well as some more advanced schedulers. This strategy should lead to a real autonomicity of components. It seems a prerequisite for adaptivity and autonomicity as discussed in Section V.

Both strategies have been validated through prototypes, the first in ProActive/GCM, the second in ADAGE [28] and GEA [17]. They run on top of various environments, from cluster like environments (ssh, batch, etc) to grid environments such as Globus.

IV Supporting M to N Communications

To meet the specific requirements and conditions of Grid computing for multiway communications, *multicast* and *gathercast* interfaces give the possibility to *manage a group of interfaces as a single entity*, and *expose* the collective nature of a given interface. Multicast interfaces allow to distribute method invocation and their parameters to a group of destinations, whereas, symmetrically, gathercast allow to synchronise a set of method invocations toward the same destination. Solutions to the problem of data distribution have been proposed within PaCO++/GridCCM [18]; these solutions can be seen as complementary to the basic distribution policy specified in this section.

IV.1 Collective Interfaces

In pure Fractal, collective bindings could be performed using composite bindings[1], which would accept one input and a collection of output, or a collection of input and one output. Collective interfaces allow GCM components to perform operations collectively on a set of components without relying on intermediate components. The objective is to simplify the design of component-based applications, and ensure type compatibility

[1] In Fractal, a composite binding is a communication path composed of a set of primitive bindings and binding components.

in a direct manner. Of course, the model still allows for the use of explicit binding components, in case of specific requirements for inter-component communications, for instance when binding interfaces of incompatible types. Though the alternative relying on composite binding could have a similar behaviour to the collective interfaces, we consider collective interfaces better adapted to the GCM as explained below.

First, we think that, for design purposes, the collective nature of the connection should be attached to the definition of the component not to its binding. This also allows control of the collective behaviour at the level of the component containing the interface, not in an external component.

Second, suppose collective interfaces would be implemented by additional components, possibly belonging to composite bindings. As in GCM the component is the unit of distribution, the question of the localisation of the additional components implementing the collective behaviour arises. The best choice would probably be to allocate the binding at the same place as one of the functional component they bind, depending on the nature of the interface; in the GCM this choice is made clear by the design of the collective interfaces. Moreover, if such binding components would be distributed, they would need to be instrumented with remote communication capabilities which would make them bigger and less efficient than collective interfaces.

Here again, the granularity of the envisioned component model plays a crucial role: making the component the unit of distribution and mobility requires primitive components to encapsulate code for managing those aspects. This makes such components inadequate for encoding basic features like collective interfaces. Indeed, it would be inefficient to attach to interfaces, or to composite binding implementing collective communication, the code necessary to manage local threads and mobility, for example.

Preliminary Remark 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*.

The notion of collective interface is not linked to any communication semantics: communication between components can be implemented for example by message passing, remote procedure calls, or streaming. However we present the particular case of remote method invocations in the remaining of this section, because of its richer implications on typing of interfaces and on the component composition. Experiments on the implementation of collective communications for components interacting by asynchronous remote method invocations have been conducted over the ProActive middleware, and proved to be quite efficient and convenient to program distributed applications [7]. However, the notions of multicast and gathercast interfaces are clearly also adapted to other communication semantics, the consequence on type compatibility between interfaces can be inferred from the case presented in this section.

IV.2 Multicast Interfaces: 1 to N Communications

Multicast interfaces provide abstractions for one-to-many communication. First, we will define this kind of interfaces, next we will detail the needed update for interfaces signature and at the end of this section we will address the distribution of parameters and invocations.

Multicast interfaces can either be used internally to a component to dispatch an invocation received by the components to several of its sub-entities, or externally to dispatch invocations emitted by the component to several clients.

IV.2.1 Definitions

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

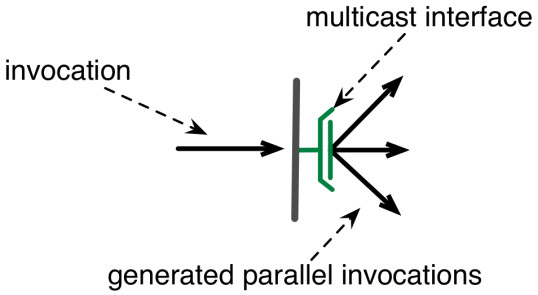


Fig. 2 Multicast interfaces

A single invocation on a multicast interface is transformed into a set of invocations. These invocations are forwarded to a set of connected server interfaces (Fig. 2). The semantics concerning the propagation of the invocation and the distribution of parameters are customisable. The result of an invocation on a multicast interface - if there is a result - is a list of results. Invocations on 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*.

For example, in a composite component, a *multicast internal client interface* transforms each single invocation into a set of invocations that are forwarded to bound server interfaces of inner components.

To support multicast interfaces, we need to extend the type system of Fractal by adding the `String getFcItfCardinality ()` method to the InterfaceType interface. The interface type is extended for dealing with new cardinalities: the `getFcItfCardinality()` method returns a string element, which is convenient when dealing with more than two kinds of cardinalities. The type factory method `createFcItfType` is extended with the `String cardinality` parameter.

The BindingController also needs an extension to support only removing of some bound interface: `void unbindFcMulticast(String name, Object itf)`.

This specification does not make any assumption about the communication paradigm used to implement the multicast invocations ([31, 35]).

IV.2.2 Automatic Data distribution

The signature of multicast interface can be different from the single interfaces it is bound to. We detail this typing issue and its relation with data distribution in this

section and the following. This section focuses on a simple view where the parameters that are to be distributed are lists, and thus the distribution can be performed automatically: lists are distributed element-wise, and other elements are kept as non-splittable. Consequently, we provide in this section two basic distribution policies for parameters: *broadcast* consists in sending the same parameters to each of the connected server interfaces; and *scatter* is only available for lists, it strips the parameter so that the bound components will work on different data.

Returned Result For each method invoked and returning a result of type `T`, a multicast invocation returns an aggregation of the results: a `list<T>`.

For instance, consider the signature of a server interface:

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

A multicast interface may be connected to the server interface with the above signature only if its signature is the following (recall that `List<A>` can be any type storing a collection of elements of type `A`):

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

In that case, we say that `I` is the type of the multicast interface *on the server side*, i.e., the type of the server interfaces the multicast can be bound to, and `J` is the type *on the client side*, i.e. the type of the mutlicast interface itself.

Where to define multicast interfaces? Collective interfaces are defined in the ADL; two new *cardinalities* – multicast and gathercast – has been added to fractal specification. The cardinality of an interface can be single, collection, *multicast*, or *gathercast*.

Where to specify parameters distribution? The ADL files aren't the right place to specify the parameter distribution because distribution is too much dependent on the implementation. Thus, the best place to specify distribution policy is inside the interface definition, e.g. using annotations in the case of Java. In addition, we propose to specify and modify the distribution policy in a dedicated controller, named CollectiveInterfacesController. The policy for managing the interface is specified as a construction parameter of the CollectiveInterfacesController. This policy is implementation-specific, and a different policy may be specified for each collective interface of the component.

How to specify the distribution of parameters into a set of invocations? Remember we focus on two possible data distribution basic policies: broadcast and scatter. In the broadcast mode, all parameters are sent without transformation to each receiver. In the scatter mode however, many configurations are possible, depending upon the number of parameters that are lists and the number of members of these lists. In the automatic distribution policies, parameters to be scattered are of type `list<T>` on the

client side, and of type `T` on the server side. Parameters to be broadcasted must be of the same type on the client and on the server side.

A general solution in the case of a single parameter to be distributed is to perform as many invocations as there are elements in the list.

When several parameters are to be distributed, there is not a single general solution. We propose to define, as part of the distribution policy, the multiset[2] F of the combination of parameters, where each element $f_j \in F$ is such that, $f_j \in [1..k_1] \times [1..k_2] \times .. \times [1..k_n]$, where n is the number of formal parameters of the invoked method which are to be scattered, and $k_i, 1 \leq i \leq n$ the number of values for each scattered actual parameter. This multiset allows the expression of all the possible distributions of scattered parameters, including Cartesian product, and one-to-one association. The cardinal of F also gives the number of invocations which are generated, and which depends on the configuration of the distribution of the parameters.

As an illustrative example, the Cartesian product of n parameters is expressed as follows:

$$\{(i_1, \ldots, i_n) | \forall l \in [1..n],\ i_l \in [1..k_l]\}$$

One-to-one association is expressed as follows when $k_1 = k_2 = \ldots k_n$:

$$\{(i, \ldots, i) | i \in [1..k]\}$$

The number of occurrence in the multiset is useful when several identical calls have to be produced, e.g. to duplicate the computation in order to tolerate the failure of some of the clients.

To summarise, for automatic data distribution in multicast interfaces:

- If the return type of the function is `T` on the server side it must be `list<T>` on the client side.
- For each parameter: if the type of the parameter is `list<T>` on the client side and `T` on the server side, then this parameter is scattered, the combination of scatter modes is defined by an external function; else, the type of the parameter is `T` on both client and server side, the parameter is broadcasted.

IV.2.3 Defining Complex Distribution Policies

This section releases the strict constraints on typing for multicast interfaces given in the preceding section, by relying on user-defined distribution or aggregation functions, and involving constraints on the resulting typing of multicast interfaces. In the general case, distribution policies may depend on the number of bound components, but for simplicity we will not explicitly use this parameter in this section. The constraints specified in this section should be used when type checking the bindings between components involved in the multicast interface.

Aggregating Results The constraint of having lists as results for multicast invocations may be relaxed by providing an aggregation mechanism that performs a reduction. Until now, we have defined a basic aggregation function which is concatenation, but any function can be used for aggregating results, leading to the following typing constraint (relate to Figure 3 for name convention):

[2] A multiset is a set where the number of occurrences of each element matters.

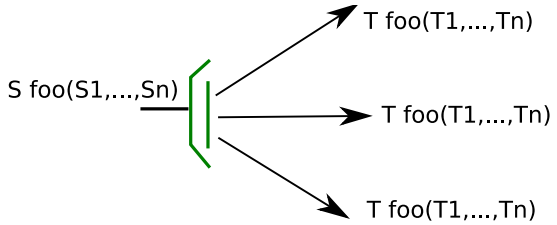


Fig. 3 General case of type conversion through a multicast interface

> If the returned type of the multicast interface is of type `S`, on the left side (i.e., if `S` is the type of the client interface), and of type `T`, on the right side (i.e., if`T` is the type of the server interfaces the multicast is connected to) then the multicast interface should be attached an aggregation function of type:
>
> $$\texttt{List<T>} \rightarrow \texttt{S}$$

Section IV.2.2 discussed the simplest case where `S = List<T>` and the aggregation function is the identity.

Depending on the operations performed by this last function, it might be necessary to synchronise the achievement of the different calls dispatched by the multicast operation. For example, it is impossible to return the maximum of all results before waiting for all of them to be arrived at the multicast interface.

Here are a few examples illustrating different possible aggregations of results for a multicast interface:

- The result is the sum of the results computed for each of the n calls distributed to the destination components:
  n integers are summed into one integer; the signature of the aggregation function is: `List <int>` $\rightarrow$ `int`. The multicast interface has the return type: `int`.
- The multicast interface returns the result given by the majority of calls.
  n results are reduced to a single one plus an occurrence count. The signature of the aggregation function becomes: `List<T>` $\rightarrow$ (`T`, `int`). The multicast interface returns a `(T,int)`.
- n pieces of an array are gathered into one single array to be returned.
  The signature of the aggregation function is: `List <Array<A> >` $\rightarrow$ `Array<A>`. The multicast interface has the return type: `Array<A>`.

Distributing Parameters This generalisation could also be applied to the distribution of invocation parameters. In Section IV.2.2, if an argument of a call toward a multicast interface is of type `S`, then the type of the argument received on one of the bound interfaces is either `S` (argument broadcasted as it is), or `T` if `S` is of the form `List<T>`. More generally, we can have any transformation of arguments type through the multicast interface:

> If the arguments of the multicast interface (i.e., the parameters of the call) are of type `Si`, $1 \leq i \leq n$ on the client side (left part of Figure 3), and of type `Ti`, $1 \leq i \leq n$ on the server side (right part of Figure 3) then the multicast interface

should be attached a distribution function returning a list of parameter sets to be sent, its type should be:

$$\texttt{S1..Sn} \rightarrow \texttt{List} <(\texttt{T1}, .., \texttt{Tn})>$$

We provide a few examples illustrating different possible type conversions for arguments of a multicast interface (the last two being the ones already presented in Section IV.2.2):

- Blocks of an array to be dispatched differently depending on the number of destination components in parallel (N):
  One call with parameter of type `Array<A>` becomes N calls with parameter of type `Array<A>` containing pieces of the original array. Distribution function is of type: `Array<A>` $\rightarrow$ `List< Array<A> >`.
- Scatter:
  One call with parameter of type `List<A>` becomes *length*(`List` $<$ `A` $>$) calls with parameter of type `A`. Distribution function is of type: `List<A>` $\rightarrow$ `List<A>`.
- Broadcast: same invocation replicated to N components in parallel:
  One call with parameter of type `A` becomes N calls with parameter of type `A`. Distribution function is of type: `A` $\rightarrow$ `List<A>`.

IV.2.4 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 to the connected server interfaces? This is determined by a function, which knowing the number of server interfaces bound to the multicast interface and the list of invocations to be performed, describes the dispatch of the invocations to those interfaces.

Consider the common case where the invocations can be distributed regardless of which component will process the invocation. Then a given component can receive several invocations; it is also possible to select only some of the bound components to participate in the multicast. In addition, this framework allows us to express naturally the case where each of the connected interfaces has to receive exactly one invocation, in a deterministic way.

IV.3 Gathercast interfaces: M to 1 Communications

Gathercast interfaces provide abstractions for many-to-one communications. Gathercast and multicast interface definitions and behaviours are symmetrical [4].

Gathercast interfaces can either be used internally to a component to gather the results of several computations performed by several sub-entities of the component, or externally to gather and synchronise several invocations made toward the component.

IV.3.1 Definition

A gathercast interface transforms a set of invocations into a single invocation.

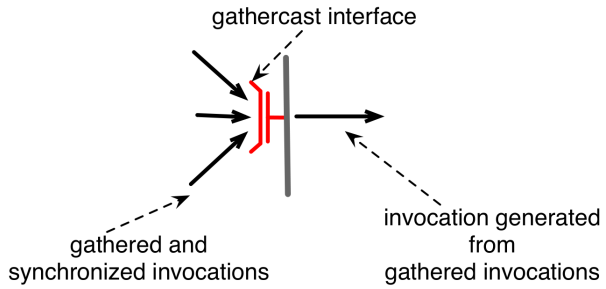


Fig. 4 Gathercast interface

Gathercast interfaces gather invocations from multiple source components (Figure 4). A gathercast interface coordinates incoming invocations before continuing the invocation flow: it may define synchronisation barriers and may gather incoming data. Return values are redistributed to the invoking components.

For example, in a composite component, a *gathercast internal server interface* transforms a set of invocations coming from client interfaces of inner components into a single invocation from the component to the external world.

For synchronisation purposes, gathering operations require knowledge of the participants (i.e. the clients of the gathercast interface) in the collective communication. As a consequence, *bindings to gathercast interfaces are bidirectional links*, in other words: a gathercast interface is aware of which interfaces are bound to it, this should be realized by the binding mechanism.

IV.3.2 Synchronisation operations

Gathercast interfaces provide one type of synchronisation operation, namely message-based synchronisation capabilities: the message flow can be blocked upon user-defined message-based conditions. *Synchronisation 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 synchronisations, based on the content and number of the messages, or based on temporal conditions, and it is possible to combine these different kinds of synchronisations.

IV.3.3 Automatic Data aggregation and Redistribution

This section details the parameter gathering and result redistribution that can be performed automatically by a gathercast interface.

Gathering parameters The gathercast interface aggregates parameters from method invocations. Thus the parameters of an invocation coming from a gathercast interface are actually lists of parameters. If, on the client side, invocations are on the form `void foo(T)` then the generated invocations necessarily have the type `void foo(list<T>)` on the server side. In other words, if the client interfaces connected to the gathercast are of type `void foo(T)`, then the gathercast (server) interface itself is of type `void foo(list<T>)`.

Redistributing results The distribution of results for gathercast interfaces is symmetrical with the distribution of parameters for multicast interfaces, and raises the question: where and how to specify the redistribution?

The place where the redistribution of results is specified is similar to the case of multicast interfaces: the redistribution is configured through metadata information for the gathercast interface. This could for example be specified through annotations, or be inferred from the type of the interface.

The way redistribution is performed is also similar to multicast interfaces. It also necessitates a comparison between the client interface type and the gathered interface type. If the return type of the invoked method in the client interfaces is of type `T` and the return type of the bound server interface is `List<T>` then results can be scattered: each component participating to the gather operation receives a single result (provided the result is a list of the same length as the number of participants). Otherwise, results should be broadcasted to all the invokers and the return type must be *identical* on the client and the server side. A redistribution function can also be defined as part of the distribution policy of the gathercast interface, it is configurable through its collective interface controller.

IV.3.4 Defining complex distribution Policies

The symmetric of multicast interfaces general specification can be defined for redistribution of results for gathercast interfaces, and aggregation of parameters of calls toward a gathercast interface. For example, the constraint of having lists as parameters for gathercast invocations may be relaxed by providing a reduction function, and verifying at connection type the type compatibility between the reduction function and the bound interfaces.

IV.4 The MxN Problem

The support of parallel components raises the concern of efficient communications between them. This section focuses on the MxN problem, i.e., efficient communication and exchange of data between two parallel programs, consisting respectively of M and N entities. In the GCM, such a pattern can be straightforwardly realized by binding a parallel component with a gathercast internal server interface to a component with a multicast internal client interface. However, efficient communications between two parallel components requires direct binding so as to support direct communications between the involved inner components on both sides; this mechanism is called MxN communications. End users expect to have MxN communications to provide performance scalability with the parallelism degree. Whereas Sections IV.4.1 to IV.4.3 focus on data distribution and establishment of bindings, Section IV.4.4 discusses synchronisation of such parallel components.

IV.4.1 Principles

A naive and not optimised solution for MxN coupling is shown in Figure 5. The respective output of the M inner components is gathered by the gathercast interface; then this result is sent as it is to the multicast interface; finally, the message is scattered to

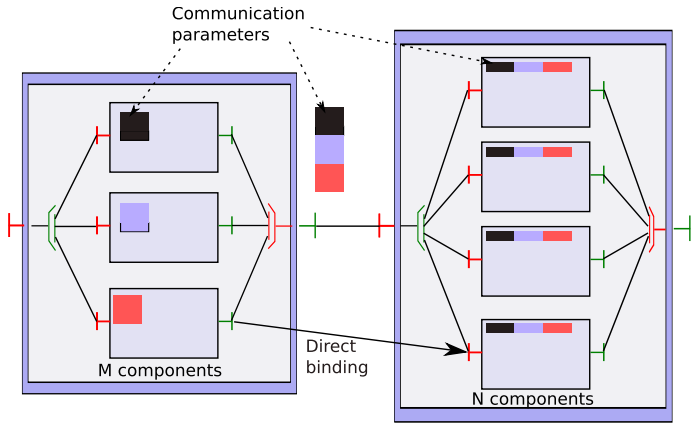


Fig. 5 Gathercast to multicast

the N inner components connected to the multicast interface, so data is redistributed by the multicast interface.

Obviously, this naive solution creates a bottleneck both in the gathercast and in the multicast interfaces. Efficient communications require some forms of direct bindings between the inner components according to the redistribution pattern, like shown by the arrow of Figure 5 drawn between an arbitrarily chosen pair of inner components from both sides. In the general case, implementing such direct bindings require to replace the couple gathercast + multicast interfaces by M multicast interfaces plus N gathercast interfaces. Each inner component on the left hand side is responsible for sending its own data to all the concerned components; on the right hand side, each inner component is responsible for gathering the messages it receives, and performing its piece of the global synchronisation. This creation of additional collective interfaces avoids the bottleneck occurring in the single gathercast or multicast interface. We show below how such an optimisation can be implemented in the case of a specific but classic scenario.

IV.4.2 Example of a Direct Binding

This section illustrates a way to ensure the M-by-N optimisation in a particular case that is relatively frequent. It both illustrates the possibility for multicast and gathercast interfaces to enable optimised communications, and it shows the necessity for highly parameterisable collective interface. Indeed optimised communication patterns are simply performed by connecting additional gathercast and multicast interfaces, parameterised depending on the data distribution and the topology.

Suppose two composites CM and CN composed of M and N components respectively, must exchange data by blocks, as shown in Figure 6. For the sake of simplicity we suppose that each of the M inner components send data of size d and each of the N components must receive data of size d' ($M \times d = N \times d'$).

We denote M_i, $0 \leq i < M$ the inner components of CM, and symmetrically, N_j, $0 \leq j < N$ the inner components of CN. Consequently, considering the data to be exchanged as an interval of size $d \times M = d' \times N$, each component exchanges the data

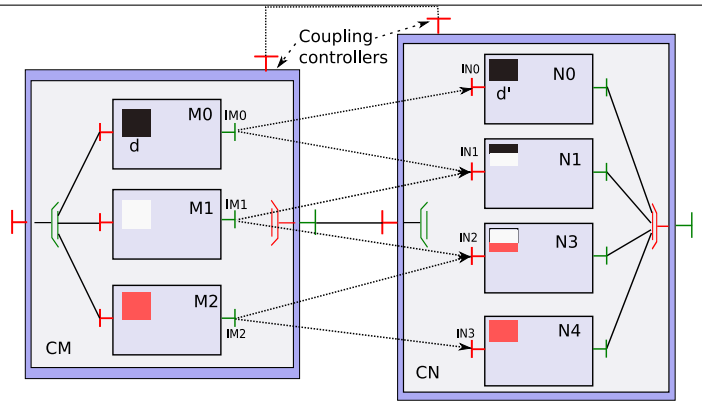


Fig. 6 Communications resulting from an MxN direct binding

in the following range:

$$M_i \text{ produces } [d \times i, d \times (i+1)[\qquad N_j \text{ consumes } [d' \times j, d' \times (j+1)[$$

Bindings Each of the M_i components will have its client interface turned into a multicast client interface with the same signature (called IM_i). Symmetrically, each of the N_j components will have its server interface turned into a gathercast server interface (called IN_j). The direct bindings that must occur should ensure the direct communication between components having to transmit data. Components are connected if there is an intersection between the range of data sent by one and the range that must be received by the other. Bindings are formally defined as follows: IM_i is to be bound to IN_j iff $\exists l \in [d \times i, d \times (i+1)[$ s.t. $l \in [d' \times j, d' \times (j+1)[$. In a more constructive manner, one can specify the indices of the client components:

$$IM_i \text{ must be bound to all the } IN_j \text{ s.t. } ((d/d') \times i) - 1 < j < (d/d') \times (i+1)$$

Communications We define now what elements are directly sent from one inner component of CM to the inner components of CN. Each M_i has to send to N_j the elements in the global range:

$$[d \times i, d \times (i+1)[\cap [d' \times j, d' \times (j+1)[$$

which is necessarily non-empty if M_i is connected to N_j. This set represent logically the intersection between the produced data and the consumed one.

IV.4.3 Using Controllers to set up MxN Bindings

This section defines a possible configuration phase for coupling two parallel components, in a MxN manner in a very general setting. It relies on the existence of controllers (called coupling controllers, that could be associated to collective controllers) both at the level of parallel components (Figure 6) and at the level of the inner ones.

When binding two parallel components, both coupling controllers exchange information about their respective collective interfaces (cardinality, data distribution pattern, size and type of data, ...), and the reference of internal components attached to this collective port. Relevant information is then passed to the coupling controllers

of the inner components so as to configure them correctly. Once configured, the direct communication (data redistribution and synchronisation) is straightforward: every inner component is aware of the components it communicates with, as well as data distribution information.

This controller-based approach is suitable to implement the redistribution described in the example above (Section IV.4.2). In this case, controllers just have to exchange the cardinality of their respective interfaces (M and N), and the references to the inner components. Controllers create and configure interfaces in the inner components accordingly to the formulas of Section IV.4.2.

IV.4.4 Synchronisation Issues

Additionally to the data redistribution, the gathercast-multicast composition plays a synchronisation role. Indeed, in Figure 5 thanks to the gathercast interface the computation can only start on the right hand side when all the inner components in the left hand side have sent their output. The introduction of the gathercast interfaces in the right hand side (Figure 6) moves this synchronisation behaviour to the N inner components. If the MxN direct communication pattern is such that each of the N processes receives data from all the M processes, then the behaviour of the system with direct bindings is equivalent to the original one. Else performing the same synchronisation as the not optimised version requires all the clients to send a message to all the gathercast interfaces, some of them being only synchronisation signals. However, global synchronisation is not required by all the applications, and in this case more optimisation is possible: if only data redistribution is important, then only bindings for transmitting data must be carried out.

IV.5 Collective Interfaces and Hierarchy

Let us conclude this section by a study on the influence of hierarchy on the notion of collective interfaces.

Basically, the existence of composite components entails the existence of internal interfaces, allowing collective interfaces to act internally to a component and collectively on the content of a composite component.

Except from this, the existence of collective interfaces is not related to hierarchy at the level of the component model. However at the applicative level, composition of hierarchy and collective operation allows the programmer to easily design complex component systems, like hierarchical master-slave for example. To summarise, the impact of collective interfaces associated with hierarchy is that any component of a system can be considered as, and transformed into, a parallel component in a very natural manner.

V Adaptivity and Autonomicity

To provide dynamic behaviour of the component control, we propose to make it possible to consider a controller as a sub-component, which can then be plugged or unplugged dynamically. As in [37], we promote the idea to adopt a component-oriented approach to express the control part of a component. On the contrary, in the Julia Fractal

implementation for instance, control part is expressed in an object-oriented fashion. Adaptivity of a component in an open and large-scale system as a computing grid can be a complex task to orchestrate and implement. So, relying on a component-oriented approach for the control part can ease its design and implementation, thus increasing the component adaptation ability.

Additionally, autonomicity is the ability for a component to adapt to situations, without relying on the outside. Several levels of autonomicity can be implemented by an autonomic system of components. The GCM defines four autonomic aspects, and it gives a precise interface for each of these four aspects. These interfaces are non-functional and exposed by each component.

V.1 A Refinement of Fractal for Non-Functional Adaptivity

In component models as Fractal, or Accord [29] for example, adaptation mechanisms are triggered by the control, also named non-functional (NF), part of the components. This NF part, called the *membrane* in Fractal and GCM, is composed of *controllers* that implement NF concerns. Interactions with execution environments may require complex relationships between controllers. Examples of use-cases include changing communication protocols, updating security policies, or taking into account new runtime environments in case of (mobile) components running on mobile devices interconnected to the core computing grid.

In this section we focus on the adaptability of the *membrane*. Adaptability means that evolutions of the execution environment have to be detected and acted upon; this process may imply interactions with the environment and with other components. Our purpose in the GCM definition with respect to adaptivity is not to provide adaptive algorithms but to offer the support for implementing them as part of the control part of the components, and even more the possibility to plug dynamically different control strategies i.e. to adapt the control part itself to the changing context.

In the GCM, we want to provide tools for adapting controllers. This means that these tools have to manage (re)configuration of controllers inside the membrane and the interactions of the membrane with membranes of other components. For this, we provide a model and an implementation, applying a component-oriented approach for both the application (functional) and the control (NF) levels. Applying a component-oriented approach to the non-functional aspects allows them to feature structure, hierarchy and encapsulation. The same way has been followed or advocated in [36,37,32, 26].

The solution adopted in the GCM is to allow, like [37,32], the design of the membrane as a set of components that can be reconfigured [14]. [8] goes more into details and describes a structure for the composition of the membrane, its relationships with the content of the component, and an API for manipulating it. Note that it does not seem reasonable to implement, like in AOKell, the membrane as a composite GCM component: due to the distributed nature of GCM (implying that a GCM component would in general involve a much higher overhead than a Fractal one), having to cross an additional composite component boundary to switch into or from the control part of a GCM component would involve a sensible overhead. So we came to the idea of having the component-based system defining the non-functional features be totally diluted in the membrane of the component containing the functional code (called *host* component in this case).

In order to be able to compose non-functional aspects, the GCM requires the NF interfaces to share the same specification as the functional ones: role, cardinality, and contingency. For example, comparatively to Fractal, the GCM adds *client non-functional interfaces* to allow for the composition of non-functional aspects, reconfigurations and component re-assembling at the non-functional level.

To summarise, the GCM is provided with the possibility to implement as components (part of) the membrane and thus benefit from strong component structure and reconfiguration capabilities.

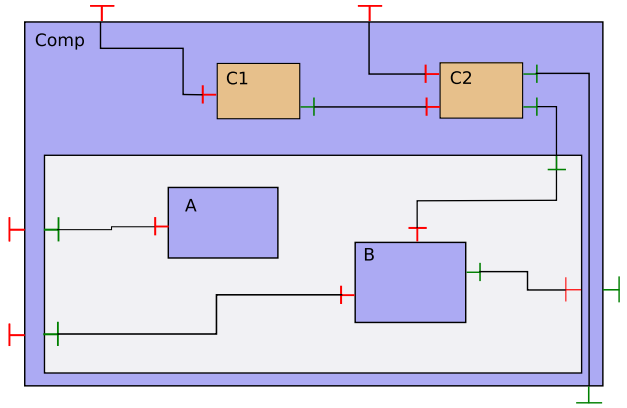


Fig. 7 A composite with pluggable controllers

A Small Example Figure 7 illustrates the structure of the membrane using components. In the figure, two non-functional components are assembled in the component's membrane, but more importantly, the membrane can rely on client non-functional interfaces, both internal to allow connection to inner components, and external to allow connections with other components, dedicated to the management and monitoring of the application for example. This both gives a structure to the non-functional concerns of the component *Comp*, but also allows the reconfiguration at the non-functional level, in order to adapt it to the changes in the environment.

Life-cycle Issue This new structure for controllers raises the following question: "What is the life-cycle of a component used inside the membrane?" In Fractal, invocation on controller interfaces *must* be enabled when a component is (functionally) *stopped*, and obviously for changing the bindings of a component, this component must be *stopped*. In other words, a controller of a Fractal component is an entity that does not follow the classical life-cycle of a component, in particular, it can never enter a stop state. Consequently, GCM components can not adhere to this specification, otherwise, their membrane could not be reconfigured.

The solution we propose consists in a more complex life-cycle for component controllers, allowing to separate partially the life-cycle states of the membrane and of the content. When a component is *functionally stopped* (which corresponds to the stopped state of the Fractal specification), invocation on controller interfaces are enabled and

the content of the component can be reconfigured. Whereas, when a component is *stopped*, only the controllers necessary for configuration are still active (mainly binding, content, and life-cycle controllers), and the other components in the membrane can be reconfigured.

Thanks to the new component architecture defined, two kinds of reconfigurations are possible: reconfiguration of the functional inner component system, following the idea of hierarchical autonomic decision paths [1]; and reconfiguration of the membrane itself when the adaptation is done along the NF properties of the host component.

V.2 Autonomic Components

GCM supports all the mechanisms needed to implement autonomic components, as stated in the previous sections. In particular, the availability of membrane components as well as the possibility to hierarchically compose new components from simpler, existing ones, can both be exploited to support different autonomic management features. More in detail, two distinct kinds of autonomic management are considered as *first class citizens* in GCM:

- The one taking care of simply adapting the single component to the changing conditions of the component"external" environment; a notable example could be the one wrapping component interaction mechanisms in such a way that the interactions can be performed using *secure* communication mechanisms rather than insecure ones. This is the kind of self-configuring, adaptation autonomic behaviour expected from components aware of the fact they can live in secure or insecure frameworks.
- The one taking care of adapting the component internal behaviour to match external, non functional requirements; a notable example could be the one adjusting the parallelism degree of a parallel composite component in such a way that a non functional performance contract is kept satisfied during the execution of the composite component activities. This is again a kind of self-configuring and self-healing autonomic behaviour [27].

In order to provide autonomic component management, GCM programming model supplies two different facilities to the GCM user/programmer. On the one hand, GCM provides all those mechanisms needed to implement the autonomic managers. These mechanisms include the ability to implement membrane as components discussed in the previous section. But they also include some lower level mechanisms that can be used to "sense" both the component execution environment and some component internal features of interest for the autonomic management. As an example, mechanisms are provided to "introspect" features of the component related to its implementation. An autonomic manager supposed to control component performance must be enabled to test component response/service time, for instance. Therefore some mechanisms are supplied within GCM that allow to probe such values. The set of mechanisms of this type provided to the autonomic manager programmers define, *de facto*, the kind of managers implementable in the GCM framework.

On the other hand, a methodology aimed at supporting autonomic component managers is provided, such that programmers of the manager components do not have to rewrite from scratch each one of the managers included in the components. Such a methodology can be basically stated as a set of guidelines and rules to be adopted when

programming the autonomic component managers, of course. In order to be more effective, GCM also provides the autonomic manager programmers with a set of autonomic manager skeletons/design patterns that can be easily customised properly supplying the skeleton/design pattern parameters. These manager patterns capitalise on the experiences coming from the software engineering autonomic management research track as well as all the experience acquired in the algorithmic skeletons and design pattern areas.

Following this approach, the GCM autonomic manager programmer can pick up one of two ways:

- he can customise a (composition of) autonomic manager skeleton(s) by providing proper parameters and therefore he can get very rapidly a complete manager whose behaviour (*modulo* the provided parameters) has already been tested, debugged and proven correct, or
- in case the provided manager skeletons do not fit user requirements, he can go through the complete (re-)writing of a new autonomic manager, exploiting the provided API to access component internal features as well as component environment features, and implementing his own autonomic policies.

In [2] it has already been outlined how autonomic manager skeletons (called "behavioural skeletons" to distinguish them from the classical "algorithmical skeletons" that are only related to the functional computation features) can be designed in GCM that can autonomically take care of the performance issues of notable parallel component compositions. Behavioural skeletons abstract common autonomic manager features leaving the autonomic manager programmer the possibility to specialise the skeleton to implement the particular autonomic manager he has in mind. More in detail, behavioural skeletons aim to describe recurring patterns of component assemblies that can be (either statically or dynamically) equipped with correct and effective management strategies with respect to a given management goal. Behavioural skeletons help the application designer to 1) design component assemblies that can be effectively reused, and 2) cope with management complexity by providing a component with an explicit context with respect to top-down design (i.e. component nesting)

Parallelism management can be designed and parameterised in the same way as classical, functional algorithmical skeletons abstract common features of parallelism exploitation patterns leaving the programmers the possibility to model their own parallel computations by providing suitable skeleton parameters, including, in the general case, sequential code parameters completely specifying the actual computation to be performed.

Technically, because the membrane components are still under development, the behavioural skeletons discussed in [2] have been currently implemented as inner components of composite components. An implementation of behavioural skeletons based on membrane components can now be considered, it will exploit several useful membrane component features, such as the ability to implement client interfaces.

VI Summary and Conclusion

In this paper, we presented the key features of a Grid-oriented component model: the GCM. Relying on Fractal as the basic component structure, the GCM defines a set of features which are necessary to turn the Fractal model into a Grid compliant one.

GCM is more than a small extension of Fractal: it provides a new set of component composition paradigm through multicast and gathercast, addresses the issue of distributed component deployment, and provides support for autonomous components. Overall, the GCM can be considered as a component model on its own. Conformance to the GCM can be summarised as follow:

- Support for deployment of components, either relying on deployment descriptors, or featuring an automatic mapping to the infrastructure,
- Possibility to collectively compose and target sets of components: existence of multicast and gathercast interfaces,
- Support for autonomic components: possibility to design membranes as component systems, to compose (i.e. bind together) non-functional features possibly distributed over several components, and support for self-adaptation of the components to both evolving environments and evolving requirements.

GCM has been used in different settings showing the effectiveness of this approach. First, a prototype of the ProActive implementation of the GCM has already been used to build and deploy over a Grid a numerical computation application for electromagnetism [34]. Moreover, in the context of the *common component modeling example (CoCoME)* context, GCM components have been modeled and specified, and a prototype implementation has been realized [11]. The CoCoME consists of a point-of-sale example featuring distribution, asynchronism, and collective communications.

Interoperability between GCM and other standards or component models has been demonstrated, first through effective interactions between CCA and GCM components [30], and second by the possibility to expose component interfaces as web services [19].

The CoreGRID Institute on Grid Systems, Tools, and Environments has been working on a methodology for the design and implementation of a generic component-based Grid platform [13] collating the innovative efforts of a number of partners from several European countries. The research activities and results show that the GCM can be used to implement a Grid runtime environment. GCM has proved to be adequate to implement development, deployment, monitoring, and steering tools. As a result, the Grid integrated development environment and the component-based integrated toolkit, based on the GCM, provide a framework that enables rapid development of Grid applications and the transparent use of available resources at runtime. These achievement show the adequacy of the GCM for developing not only Grid applications but also a Grid runtime and environment.

The experiences mentioned above allow us to evaluate the GCM relatively to the objectives given in the introduction. First, the hierarchical aspect is really a key feature to better address scalability in practice. Second, expressiveness of the collective interfaces is generally adequate as showed by the programming of SPMD-like interactions; but the specification of distribution policies is still to be improved. Indeed, allowing the GCM implementation to reach the expressiveness of the distribution policies described in Sections IV.2.3, and IV.3.4, and thus allowing real program to express simply complex distributions is still a real challenge. Finally, we also showed the adequacy of the GCM to express autonomic adaptations [2, 6]. Thus we estimate that the GCM greatly improves expressiveness, scalability, and adaptivity of Grid applications, even if the model and its implementation are still to be improved. One difficulty that has been encountered several times is the management of distributed asynchronous components, and in particular the problem of stopping such components; however some solutions have been recently suggested for this problem [39, 25].

References

1. Aldinucci (M.), Bertolli (C.), Campa (S.), Coppola (M.), Vanneschi (M.), Zoccolo (C.). Autonomic Grid Components: the GCM Proposal and Self-optimising ASSIST Components. In *Joint Workshop on HPC Grid programming Environments and Components and Component and Framework Technology in High-Performance and Scientific Computing at HPDC'15*, 2006.
2. Aldinucci (M.), Campa (S.), Danelutto (M.), Dazzi (P.), Kilpatrick (P.), Laforenza (D.), Tonellotto (N.). Behavioural skeletons for component autonomic management on grids. In M. Danelutto, P. Frangopoulou, V. Getov, eds., *Making Grids Work*, CoreGRID, 2008. ISBN 978-0-387-78447-2.
3. Armstrong (R.), Gannon (D.), Geist (A.), Keahey (K.), Kohn (S.), McInnes (L.), Parker (S.), Smolinski (B.). Toward a common component architecture for high-performance scientific computing. In *Proceedings of the 1999 Conference on High Performance Distributed Computing*, 1999.
4. Badrinath (B.), Sudame (P.). 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.
5. Baduel (L.), Baude (F.), Caromel (D.), Contes (A.), Huet (F.), Morel (M.), Quilici (R.). *Grid Computing: Software Environments and Tools*, chap. Programming, Deploying, Composing, for the Grid. Springer-Verlag, 2006.
6. Baude (F.), Henrio (L.), Naoumenko (P.). A Component Platform for Experimenting with Autonomic Composition. In *First International Conference on Autonomic Computing and Communication Systems (Autonomics 2007)*. ACM Digital Library, 2007. Invited Paper.
7. Baude (F.), Caromel (D.), Henrio (L.), Morel (M.). Collective interfaces for distributed components. In *CCGrid 2007: IEEE International Symposium on Cluster Computing and the Grid*, (pages 599–610), 2007. ISBN 0-7695-2833-3.
8. Baude (F.), Caromel (D.), Henrio (L.), Naoumenko (P.). A flexible model and implementation of component controllers. In *CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments*, 2007.
9. Bertrand (F.), Bramley (R.), Damevski (K.B.), Kohl (J.A.), Bernholdt (D.E.), Larson (J.W.), Sussman (A.). Data redistribution and remote method invocation in parallel component architectures. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium: IPDPS*, 2005.
10. Bruneton (E.), Coupaye (T.), Leclercq (M.), Quéma (V.), Stefani (J.B.). The fractal component model and its support in java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, **36, 11-12**, 2006.
11. Cansado (A.), Caromel (D.), Henrio (L.), Madelaine (E.), Rivera (M.), Salageanu (E.). *A Specification Language for Components Implemented in GCM/ProActive*. LNCS series. Springer Verlag, 2007. To appear.
12. CCA forum. The Common Component Architecture (CCA) Forum home page, 2005. Http://www.cca-forum.org/.
13. CoreGRID Institute on Grid Systems, Tools, and Environments. Design methodology of the generic component-based grid platform, 2008. Deliverable D.STE.07.
14. CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed), 2006. Deliverable D.PM.04, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.
15. Coulson (G.), Grace (P.), Blair (G.), Mathy (L.), Duce (D.), Cooper (C.), Yeung (W.K.), Cai (W.). Towards a component-based middleware framework for configurable and reconfigurable grid computing. *wetice*, **00**, 291–296, 2004. ISSN 1524-4547. doi:http://doi.ieeecomputersociety.org/10.1109/ENABL.2004.69.
16. Czajkowski (K.), Kesselman (C.), Fitzgerald (S.), Foster (I.). Grid information services for distributed resource sharing. *hpdc*, **00**, 0181, 2001. doi:http://doi.ieeecomputersociety.org/10.1109/HPDC.2001.945188.
17. Danelutto (M.), Vanneschi (M.), Zoccolo (C.), Tonellotto (N.), Orlando (S.), Baraglia (R.), Fagni (T.), Laforenza (D.), Paccosi (A.). HPC application execution on GRIDs. *Future Generation Grids*, (pages 263–282), 2005.
18. Denis (A.), Perez (C.), Priol (T.), Ribes (A.). Bringing high performance to the corba component model. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2004.

19. Dűnnweber (J.), Baude (F.), Legrand (V.), Parlavantzas (N.), Gorlatch (S.). *Invited papers from the 1st CoreGRID Integration Workshop, Pisa, Novembre 2005*, chap. Towards Automatic Creation of Web Services for Grid Component Composition. volume 4 of CoreGRID series. Springer, 2006.
20. Furmento (N.), Hau (J.), W.Lee, Newhouse (S.), Darlington (J.). Implementations of a Service-Oriented Architecture on Top of Jini, JXTA and OGSI. In *AxGrids 2004*, no. 3165 in LNCS, (page 90Ð99). Springer, 2004.
21. Furmento (N.), Lee (W.), Mayer (A.), Newhouse (S.), Darlington (J.). Iceni: An open grid service architecture implemented with jini. *sc*, **00**, 37, 2002. ISSN 1063-9535. doi:http://doi.ieeecomputersociety.org/10.1109/SC.2002.10027.
22. Gannon (D.). Programming the grid: Distributed software components, 2002. URL `citeseer.ist.psu.edu/gannon02programming.html`.
23. Gannon (D.), Fox (G.). Workflow in grid systems meeting. *Concurrency & Computation: Practice & Experience*, **18, 10**, 2006. Based on GGF10 Berlin Meeting.
24. Getov (V.), von Laszewski (G.), Philippsen (M.), Foster (I.). Multiparadigm communications in java for grid computing. *Commun. ACM*, **44, 10**, 118–125, 2001. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/383845.383872.
25. Henrio (L.), Rivera (M.). Stopping safely hierarchical distributed components. In *Proceedingsof the Workshop on Component-Based High Performance Computing (CBHPC'08)*, 2008.
26. Herault (C.), Nemchenko (S.), Lecomte (S.). A Component-Based Transactional Service, Including Advanced Transactional Models. In *Advanced Distributed Systems: 5th International School and Symposium, ISSADS 2005, Revised Selected Papers*, no. 3563 in LNCS, 2005.
27. Kephart (J.O.), Chess (D.M.). The vision of autonomic computing. *Computer*, **36, 1**, 41–50, 2003. ISSN 0018-9162. doi:http://dx.doi.org/10.1109/MC.2003.1160055.
28. Lacour (S.), Pérez (C.), Priol (T.). Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM International Workshop on Grid Computing (Grid2005)*. Springer-Verlag, Seattle, WA, USA, 2005.
29. Liu (H.), Parashar (M.). A Component Based Programming Framework for Autonomic Applications. In *1st IEE Int. Conference on Autonomic Computing (ICAC)*, 2004.
30. Malawski (M.), Bubak (M.), Baude (F.), Caromel (D.), Henrio (L.), Morel (M.). Interoperability of grid component models: GCM and CCA case study. In *CoreGRID Symposium in conjunction with Euro-Par 2007*, CoreGRID series. Springer Verlag, 2007.
31. Mayer (A.), Mcough (S.), Gulamali (M.), Young (L.), Stanton (J.), Newhouse (S.), Darlington (J.). Meaning and behaviour in grid oriented components. In *Third International Workshop on Grid Computing, GRID*, vol. 2536 of *LNCS*, (pages 100–111), 2002.
32. Mencl (V.), Bures (T.). Microcomponent-based component controllers: A foundation for component aspects. In *APSEC*. IEEE Computer Society, 2005.
33. omg.org team. CORBA Component Model, V3.0. http://www.omg.org/technology/documents/formal/components.htm, 2005.
34. Parlavantzas (N.), Morel (M.), Getov (V.), Baude (F.), Caromel (D.). Performance and scalability of a component-based grid application. In *9th Int. Worshop on Java for Parallel and Distributed Computing, in conjunction with the IEEE IPDPS conference*, 2007.
35. Partridge (C.), Menedez (T.), Milliken (W.). Host anycasting service. RFC 1546, 1993.
36. Seinturier (L.), Pessemier (N.), Coupaye (T.). AOKell: an Aspect-Oriented Implementation of the Fractal Specifications, 2005. `http://www.lifl.fr/~seinturi/aokell/javadoc/overview.html`.
37. Seinturier (L.), Pessemier (N.), Duchien (L.), Coupaye (T.). A Component Model Engineered with Components and Aspects. In *Proceedings of the 9th International SIGSOFT Symposium on Component-Based Software Engineering*, 2006.
38. Szyperski (C.). *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998. ISBN 0-201-17888-5.
39. Tejedor (E.), Badia (R.M.), Naoumenko (P.), Rivera (M.), Dalmasso (C.). Orchestrating a safe functional suspension of gcm components. In *CoreGRID Integration Workshop. Integrated Research in Grid Computing*, 2008.
40. Thome (B.), Viswanathan (V.). Enterprise Grid Alliance–Reference Model v1.0, 2005.