Project no.FP6-034442

GridCOMP

Grid programming with COMPonents : an advanced component platform for an effective invisible grid

STREP Project

Advanced Grid Technologies, Systems and Services

D.NFCF.01 – Non functional component subsystem architectural design

Due date of deliverable: May 31th, 2007

Actual submission date: June 26th 2007

Start date of project: 1 June 2006 **Duration:** 30 months

Organisation name of lead contractor for this deliverable: UNIPI

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination level		
PU	Public	PU

Keyword list: autonomic management, component controller, GMC, task farm
Responsible Partner: UNIPI

MODIFICATION CONTROL			
Version	Date	Status	Modifications made by
0	25-06-2007	Draft	Marco ALDINUCCI
1	04-07-2007	Draft	Marco ALDINUCCI
2	11-07-2007	Draft	Marco ALDINUCCI

Deliverable manager

- Marco Aldinucci, UNIPI

List of Contributors

- Sonia Campa, UNIPI
- Patrizio Dazzi, ISTI-CNR
- Nicola Tonellotto, ISTI-CNR

List of Evaluators

- Françoise Baude, INRIA
- Rajkumar Buyya, U. MELBOURNE

Executive Summary: This deliverable reports the architectural design of the non-functional component subsystem of CoreGrid Component Model (GCM) implementation. Since GCM components are defined as autonomic elements, their non-functional subsystem is basically an autonomic manager. As a result, the design of the non-functional component subsystem includes the definition of component assemblies that can automatically be managed, and the mechanisms and policies to manage them. At this end, we present behavioural skeletons for the GCM, which are an abstraction aimed at simplifying the development of GCM applications, and in particular self-managed ones. Behavioural skeletons abstract component self-management in component-based design as design patterns abstract class design in classic OO development. As here we just want to introduce the behavioural skeleton framework, emphasis is placed on general skeleton structure, their architectural design in GCM, and the general mechanisms needed to manage them rather than on their autonomic management policies. Preliminary experimental results are presented.

Contents

1 Introduction

The grid poses new challenges in terms of programmability, interoperability, code reuse and efficiency. These challenges mainly arise from a key feature that are peculiar to grid, namely uncertainty. Neither the target platforms nor their status are fixed during the application run on a grid [16]. This makes application adaptivity an essential feature in order to achieve high performance and to exploit efficiently the available resources [4].

The basic use of static adaptation covers straightforward but popular methodologies, such as *copy-paste*, and *OO inheritance*. A more advanced usage covers the case in which adaptation happens at run-time. These systems enable dynamically defined adaptation by allowing adaptations, in the form of code, scripts or rules, to be added, removed or modified at run-time [11]. Among them it is worth to distinguish the systems where all possible adaptation cases have been specified at compile time, but the conditions determining the actual adaptation at any point in time can be dynamically changed [8]. Dynamically adaptable systems rely on a clear separation of concerns between adaptation and application logic. This approach has recently gained increased impetus in the grid community, especially via its formalization in terms of the *Autonomic Computing* (AC) paradigm [17, 9, 6].

The CoreGrid Component Model (GCM) definition natively embodies the AC idea [13]. A GCM autonomic component consists of one or more managed components coupled with a single autonomic manager that controls them. To pursue its goal, the manager may trigger an adaptation of the managed components to react to a run-time change of application QoS requirements or to the platform status.

In this regard, an assembly of self-managed components implements, via their managers, a distributed algorithm that manages the entire application. Several existing programming frameworks aim to ease this task by providing a set of mechanisms to dynamically install reactive rules within autonomic managers. These rules are typically specified as a collection of `when`-*event*-`if`-*cond*-`then`-*act* clauses, where *event* is raised by the monitoring of component internal or external activity (e.g. the component server interface received a request, and the platform running a component exceeded a threshold load, respectively); *cond* is an expression over component internal attributes (e.g. component life-cycle status); *act* represents an adaptation action (e.g. create, destroy a component, wire, unwire components, notify events to another component's manager). Several programming frameworks implement variants of this general idea, including ASSIST [22, 4], AutoMate [19], SAFRAN [14], and finally the CoreGrid Component Model (GCM) [13]. The latter two are derived from a common ancestor, i.e. the Fractal hierarchical component model [18]. All the named frameworks, except SAFRAN, are targeted to distributed applications on grids.

Though such programming frameworks considerably ease the development of an autonomic application for the grid (to various degrees), they rely fully on the application programmer's expertise for the set-up of the management code, which can be quite difficult to write since it may involve the management of black-box components, and, notably, is tailored for the particular component or assembly of them. As a result, the introduction of dynamic adaptivity and self-management might enable the management of grid dynamism, and uncertainty aspects but, at the same time, decreases the component reuse potential since it further specializes components with application specific management code.

Within GridCOMP project, we proposed *behavioural skeletons* as a novel way to describe autonomic components in the GCM framework. 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 the programmer with component templates that, once instantiated, can take the part of general application management strategy spanning component assemblies in the horizontal (i.e. wiring) and the vertical (i.e. nesting) extent.

In Sec. 2 the basic design principles of GCM are reviewed. In Sec. 3 we introduce the concept of behavioural skeletons as parametric abstractions of GCM components autonomic features, while in Sec. 4 we illustrate a reduced set of behavioural skeletons commonly exploited in parallel computations. In Sec. 6 we illustrate the architecture and the implementation of the autonomic features described in the GCM that will be exploited to implement behavioural skeletons, and in Sec. 7 we present preliminary results of experiments of the farm behavioural skeleton. Finally, in Sec. 8 we summarize our work and our main results.

2 The Grid Component Model

GCM allows component interactions to take place with several distinct mechanisms. In addition to classical "RPC-like" use/provide ports (or client/server interfaces), GCM allows data, stream and event ports to be used in component interaction. Furthermore, collective interaction patterns (communication mechanisms) are also supported. The full specification of GCM can be found in [13].

GCM is therefore assumed to provide several levels of autonomic managers in components, that take care of the non-functional features of the component programs. GCM components thus have two kinds of interfaces: functional and non-functional ones. The functional interfaces host all those ports concerned with implementation of the functional features of the component. The non-functional interfaces host all those ports needed to support the component management activity in the implementation of the non-functional features, i.e. all those features contributing to the efficiency of the component in obtaining the expected (functional) results but not directly involved in result computation. Each GCM component therefore contains an *Autonomic Manager* (AM), interacting with other managers in other components via the component non-functional interfaces. The AM implements the autonomic cycle via a simple program based on the reactive rules described above. In this, the AM leverages on component controllers for the *event* monitoring and the execution of reconfiguration *actions*. In GCM, the latter controller is called the *Autonomic Behaviour Controller* (ABC). This controller exposes server-only non-functional interfaces, which can be accessed either from the AM or an external component that logically surrogates the AM strategy. According to GCM specification [13], we call *passive* a GCM component exhibiting just the ABC, whereas we call *active* a GCM component exhibiting both the ABC and the AM[1].

2.1 Describing Adaptive Applications

The architecture of a component-based application is usually described via an ADL (Architecture Description Language) text, which enumerates the components and describes their relationships via the *used-by* relationship. In a hierarchical component model, such as the GCM, the ADL describes also the *implemented-by* relationship, which represents the component nesting.

However, the ADL supplies a static vision of an application, which is not fully satisfactory for an application exhibiting autonomic behaviour since it may au-

[1]Notice a passive GCM component is not just an adaptable component, but is supposed to provide ports for component monitoring and steering.

tonomously change behaviour during its execution. Such change may be of several types:

- *Component lifecycle.* Components can be started or stopped.
- *Component relationships.* The used-by and/or implemented-by relationships among components are changed. This may involve component creation/destruction, and component wiring alteration.
- *Component attributes.* A refinement of the behaviour of some components (which does not involve structural changes) is required, usually over a predetermined parametric functionality.

In the most general case, an autonomic application may evolve along adaption steps that involve one or more changes belonging to these three classes. In this regard, the ADL just represents a snapshot of the launch time configuration.

The evolution of a component is driven by its AM, which may request management action with the AM at the next level up in order to deal with management issues it cannot solve locally. Overall, it is a part of a distributed system that cooperatively manages the entire application.

In the general case, the management code executing in the AM of a component depends both on the component's functional behaviour and the goal of the management. The AM should also be able to cooperate with other AMs, which are unknown at design time due to the nature of component-based design. Currently, programming frameworks supporting the AC paradigm (such as the ones mentioned in Sec. 1) just provide mechanisms to implement management code. This approach has several disadvantages, especially when applied to a hierarchical component model:

- The management code is difficult to develop and to test since the context in which it should work may be unknown.
- The management code is tailored to the particular instance of the managed elements (inner components), further restricting the possible component reusability.

For this reason, we believe that the "ad-hoc" approach to management code is unfit to be a cornerstone of the GCM component model.

3 Behavioural Skeletons

Behavioural skeletons aim to abstract parametric paradigms of GCM component assembly, each of them specialized to solve one or more management goals belonging to the classical AC classes, i.e. configuration, optimization, healing and protection.

Behavioural skeletons represent a specialization of algorithmic skeleton concept for component management [12]. Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. Behavioural skeletons, as algorithmic skeletons, represent patterns of parallel computations (which are expressed in GCM as graphs of components), but in addition they exploit the inherent skeleton semantics to design sound self-management schemes of parallel components.

Due to the hierarchical nature of GCM, behavioural skeletons can be identified with a composite component with no loss of generality (identifying skeletons as particular higher-order components [15]). Since component composition is defined independently from behavioural skeletons, they do not represent the exclusive means of expressing applications, but can be freely mixed with non-skeletal components. In this setting, a behavioural skeleton is a composite component that

- exposes a description of its functional behaviour;
- establishes a parametric orchestration schema of inner components;
- may carry constraints that inner components are required to comply with;
- may carry a number of pre-defined plans aiming to cope with a given self-management goal.

Behavioural skeleton usage helps designers in two main ways: The application designer benefits from a library of skeletons, each of them carrying several pre-defined, efficient self-management strategies; and, the component/application designer is provided with a framework that helps the design of new skeletons and their implementations.

The former task is achieved because (1) skeletons exhibit an explicit higher-order functional semantics, which delimits the skeleton usage and definition domain; and (2) skeletons describe parametric interaction patterns and can be designed in such a way that parameters affect non-functional behaviour but are invariant for functional behaviour.

4 A Basic Set of Behavioural Skeletons

Here we present a basic set of behavioural skeletons for the sake of exemplification. Despite their simplicity, they cover a significant set of parallel computations of common usage.

One class of behavioural skeletons springs from the idea of *functional replication*. Let us assume the skeletons in this class have two functional interfaces: a one-to-many stream server S, and a many-to-one client stream interface C (see Fig. 1). The skeleton accepts requests on the server interface; and dispatches them to a number of instances of an inner component W, which may propagate results outside the skeleton via C interface. Assume that replicas of W can safely lose the internal state between different calls. For example, the component has just a transient internal state and/or stores persistent data via an external data-base component.

Farm A stream of tasks is absorbed by a *unicast* S, each task is computed by one instance of W and sent to G, which collect tasks *from-any*. This skeleton can be equipped with a self-optimizing policy because the number of Ws can be dynamically changed in a sound way since they are stateless. The typical QoS goal is to keep a given limit (possibly dynamically changing) of served requests in a time frame. The AM just checks the average time tasks need to traverse the skeleton, and eventually reacts by creating/destroying instances of Ws, and wiring/unwiring them to/from the interfaces.

Data-Parallel A stream of tasks is absorbed by a *scatter* S; each task is split in (possibly overlapping) partitions, which are distributed to replicas of W to be computed. Results are *gathered* and assembled by G in a single item. As in the previous case, the number of Ws can be dynamically changed (between different requests) in a sound way since they are stateless. As in the previous case, the skeleton can be equipped with a self-configuration goal, i.e. resource balancing and tuning (e.g. disk space, load, memory usage), that can be achieved by changing the partition-worker mapping in S (and C, accordingly).

Active-Replication A stream of tasks is absorbed by a *broadcast* S, which sends identical copies to the Ws. Results are sent to G, which reduces them. This paradigm can be equipped with a self-healing policy because it can deal with Ws that do not answer, produce an approximate or wrong answer by means of a result reduction function (e.g. by means of averaging or voting on results).

The presented behavioural skeletons can be easily adapted to the case that S is a RPC interface. In this case, the C interface can be either a RPC interface or missing. Also, the functional replication idea can be extended to the stateful case by requiring the inner components Ws to expose suitable methods to serialize, read and write the internal state. A suitable manipulation of the serialized state enables the reconfiguration of workers (also in the data-parallel scenario [4]).

Anyway, in order to achieve self-healing goals some additional requirements on the GCM implementation level should be enforced. They are related to the implementation of the GCM mechanisms, such as the messaging system, the component membranes, and their parts (e.g. interfaces). At the level of interest, they are primitive mechanisms, in which correctness and robustness should be enforced ex-ante, at least to achieve some of the described management policies.

The process of identification of other skeletons may benefit from the work done within the software engineering community, which identified some common adaptation paradigms, such as *proxies* [20], which may be interposed between interacting components to change their interaction relationships; and dynamic *wrappers* [21]. Both of these can be used for self-protection purposes. As an example a couple of encrypting proxies can be used to secure a communication between components. Wrapping can be used to hide one or more interfaces whether a component is deployed into an untrusted platform.

4.1 Specifying Skeleton Behaviour

Autonomic management requires that, during execution of a system, components of the system are replaced by other components, typically having the same functional behaviour but exhibiting different non-functional characteristics. The application programmer must be confident about the behaviour of the replacements with respect to the original. The behavioural skeleton approach proposed supports these requirements in two key ways:

1. The use of skeletons with its inherent parametrization permits relatively easy parameter-driven variation of non-functional behaviour while maintaining functional equivalence.

2. The use of a formal or semi-formal specification to describe component behaviour gives the developer a firm basis on which to compare the properties of alternative realisations in the context of autonomic replacement.

The skeleton designer can use the description to prove rigorously (manually, at present) that a given management strategy will have predictable or no impact on functional behaviour. The quantitative description of QoS values of a component with respect to a goal, the automatic validation of management plans w.r.t. a given functional behaviour are interesting related topics, which are the subject of ongoing research. Examples of semi-formal specifications of the proposed skeletons can be found in [1, 5].

As byproduct, behavioural skeletons categorize GCM designers and programmers in three classes. They are, in increasing degree of expertise and decreasing cardinality:

- GCM users. They are supposed to use behavioural skeletons together with their pre-defined AM strategy. In many cases they should just instantiate a skeleton with inner components, and get as result a composite component exhibiting one or more self-management behaviours.

- GCM expert users. They are supposed to use behavioural skeletons overriding the AM management strategy. the personalization does not involve the ABC, thus does not need specific knowledge about GCM membrane implementation.

- GCM skeleton designers. They are supposed to introduce new behavioural skeletons or classes of them. At this end, the design and development of a brand new ABC might be required. This may involve the definition of new interfaces for the ABC, the implementation of the ABC itself together with its wiring with other controllers, and the design and wiring of new interceptors. This requires a quite deep knowledge of the particular GCM implementation.

4.2 GCM Specification and Behavioural Skeletons

In terms of the GCM specification [13], a behavioural skeleton is a particular composite component exhibiting an autonomic conformance level strictly greater than one, i.e. a component with passive or active autonomic control. The component exposes pre-defined functional and non-functional client and server interfaces according to the skeleton type; functional interfaces are usually collective and configurable. Since skeletons are fully-fledged GCM components, they can be wired and nested via standard GCM mechanisms. From the implementation viewpoint, a behavioural skeleton is a partially defined composite component, i.e. a component with placeholders, which may be used to instantiate the skeleton. As sketched in Fig. 1, there are three classes of placeholders:

1. The functional interfaces S and C that are GCM functional interfaces, which may be equipped with monitoring interceptors controllers (currently objects).

2. The AM that is a particular inner component. It includes the management plan, its goal, and exported non-functional interfaces.

3. Inner component W, implementing the functional behaviour.

The orchestration of the inner components, and thus ABC functionality, is implicitly defined by the skeleton class. In order to instantiate the skeleton, placeholders should be filled with suitable entities. Observe that just entities in the former two classes are skeleton specific.

In addition to a standard composite component, a behavioural skeleton is further characterized by a formal (or semi-formal) description of the component behaviour. This description can be attached to the ADL component definition via the standard GCM ADL hook, which can be used with any behavioural specification language. In this regard, a description based on the Orc language have been proposed within GridCOMP and CoreGrid projects [1, 5].

5 About GCM collective communications

The GCM specification includes *structured communications*, i.e. the support for many-to-one and one-to-many communications on top of both remote method invocation and data streaming. In particular, the current GCM specification [13], introduces a set of collective interfaces, so-called *multicast* and *gathercast* (see [13]).

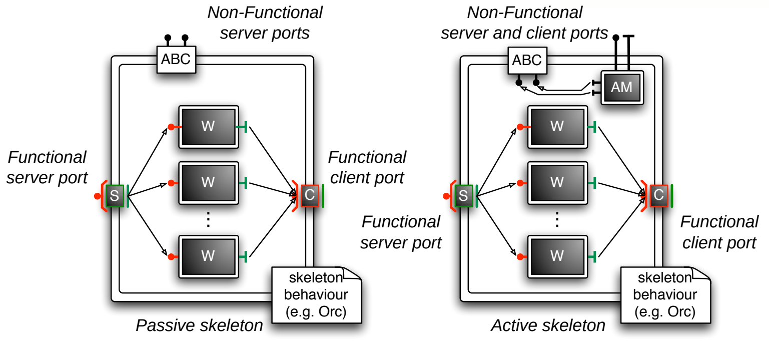

Figure 1: GCM implementation of functional replication. ABC = Autonomic Behaviour Controller, AM = Autonomic Manager, W = Worker component, S = Server interface (one-to-many communication e.g. broadcast, data-parallel scatter, unicast), C = Client interface (many-to-one communication e.g. from-any, data-parallel gather, reduce, select).

These interfaces aim to split-and-distribute/gather-and-join data in a flexible way (through the generalized aggregation mechanism). The two operations are defined as functions `T` → `list of T`, and vice-versa, and can be used with both single RPC call or stream items. However, they hardly capture typical operations performed on stream computations because they can hardly deal with many, possibly consecutive, stream items (or RPC calls) in stateful way. As an example a typical operation on streams consist in dispatching successive stream items to different components. This operation, usually called *unicast*, is naturally described by a stateful function `stream of T` → `T`, that can be hardly matched with multicast GCM type.

Within this deliverable we elaborate on the role of stream interfaces in GCM and we introduce the definition and the implementation of unicast in GCM prototype. The unicast interface is part of a set of interfaces specifically designed to cope with the distribution of consecutive stream items from a single *source_interface* toward a *target_interface* dynamically chosen in set. The interfaces in the set typically belong to different components, while the *target_interface* is dynamically chosen at the dispatch time of each item knowing the history of previous choices. As an example, this enable to dispatch consecutive items in a stream (or consecutive calls of a method) toward different components in round-robin fashion. Previously mentioned *from-any* interface covers the collection of items in a similar fashion. Several variants of this kind of interfaces can be imagined, as an example *unicast-on-demand*, *from-any-unordered*, and *from-any-ordered*. Previous works with the ASSIST coordination language proved the expressiveness and efficiency of these kind of interfaces [22, 3, 4].

6 GCM Autonomic Features Implementation

6.1 Autonomic Behavior Controller

According to the GCM specification, a component with passive autonomic behavior must expose the interfaces of some standard basic controllers (component, attribute, binding content and lifecycle controller) and a `AutonomicBehaviorController`[2], interface:

[2]The AutonomicBehaviorController was formerly named `AutonomicController`.

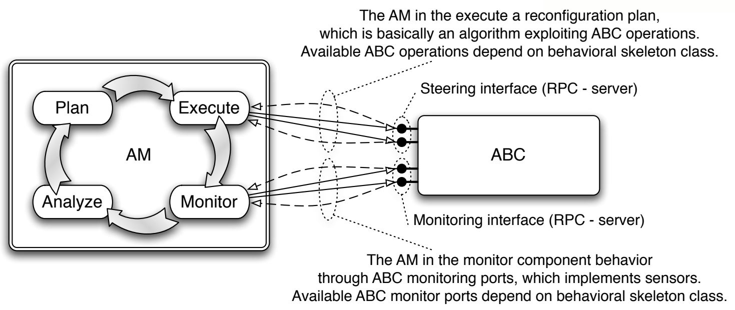
The AM in the execute a reconfiguration plan,
which is basically an algorithm exploiting ABC operations.
Available ABC operations depend on behavioral skeleton class.
Steering interface (RPC - server)
Plan
Execute
AM
Analyze
Monitor
ABC
Monitoring interface (RPC - server)
The AM in the monitor component behavior
through ABC monitoring ports, which implements sensors.
Available ABC monitor ports depend on behavioral skeleton class.

Figure 2: Relation between AM and ABC.

```
interface AutonomicBehaviorController {
    String[] listAutonomicOperations();
    any execOperation(String op, any ...);
}
```

The `listAutonomicOperations` method lists all the autonomicity-handling operations provided by the controller. Each operation returned by this method is executed by invoking the `execOperation` method, where the first argument is the identifier of the operation as returned by the `listAutonomicOperations` method and the second (vararg) argument is a list of parameters supplied to the operation.

As sketched in Fig. 2, the ABC implements the basic operations for component monitoring and steering. These operations are used within the AM to realize autonomic control strategies.

The implementation choices of ABC are targeted to the ProActive implementation of the GCM. In this deliverable, two stable implementations of this interface are provided: one for primitive components and one for composite components, and in particular for behavioural skeletons.

In the case of primitive component, the interface enables the programmers to expose primitive component monitoring and steering features. At this end, a basic implementation of the ABC for primitive component is provided. This controller, located in the component membrane, acts as a proxy for the component-specific implementation of the component monitoring and steering features. In the case of primitive components, the ABC controller consists of a controller template, since no general implementation can be given. The ABC template should be fully implemented by the component designer. The design of ABC controller is compliant with the design of other controllers, e.g. the BC for primitive components. Actually, the only required non-functional interface for this ABC implementation is the `Component` controller interface.

Besides the primitive components implementation, a general implementation of the ABC for composite components is provided for each skeleton class (e.g. functional replication). In this case, since each skeleton (class) exhibits a predetermined semantics, it is possible to provide the component designer with a fully implemented ABC controller, and then to considerably ease component designer task.

The goal of this implementation is two-fold: first, to provide an implementation of skeletons belonging to functional replication class for composite components (in particular for the farm skeleton); and to show how ABC ports can be used within AM strategies.

Given a primitive component with several server and client interfaces, the final goal is to provide a mechanism to increase its parallelism degree (and reduce it later, if necessary) in order to face sudden increases/decreases in its workload.

Increasing the parallelism of a primitive component means duplicate it and provide in some way the mechanisms needed to dispatch server interface invocations and to collect client interfaces invocations. To provide a uniform mechanism to manage the parallelism degree of a primitive component, we decided to force the component developer to wrap the original component inside a composite component with the same interfaces of the original one, the generic composite component controllers and the specific ABC controller (see Fig. 3). Being a passive GCM component, the AM is not present at this stage.

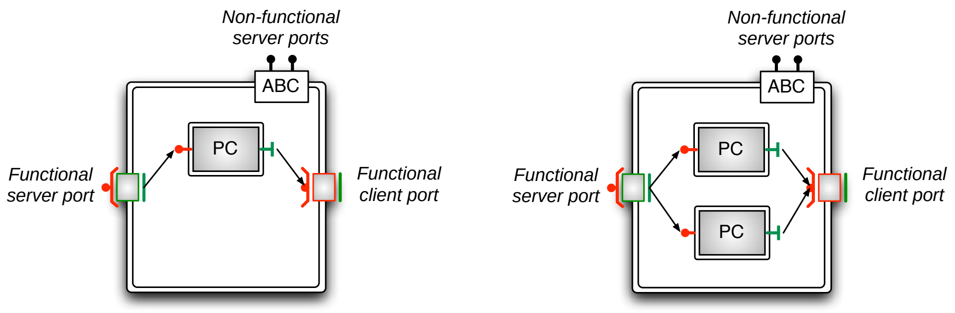

Figure 3: The wrapping of a primitive component PC and its duplication

The ABC controller exposes the server interface described above with three autonomic operations: one to query the average queueing time of the last five calls of an observed server interface and two to respectively increase and decrease the parallelism degree of the component (i.e. add/remove copies of the original primitive component). While the monitoring of the functional server interfaces can be easily implemented exploiting the interception mechanisms provided by ProActive, the run-time modification of the content of a component must be carefully planned. In the following, the implementation of the increase parallelism operation is discussed.

The addition of a primitive component to a running one requires the reconfiguration of part of the application. Two distinct scenarios are possible:

1. a primitive component must be wrapped in a composite one containing the original component and a new copy, with all the required mechanisms in the composite component membrane;

2. a new instance of a primitive component must be added to a composite one already containing several copies of the primitive component.

The first scenario corresponds to increase the parallelism degree from 1 to 2, while the second one to increase the parallelism degree from n to $n+1$ (with $n > 1$). The critical part of the whole procedure corresponds to the first scenario. Stopping a primitive component and creating a new composite component are not difficult to implement, but modifying at run-time the content of components is difficult.

In its basic form, this operation adds an exact copy of the original primitive component, inside the composite component. The sequential steps performed by this operation are depicted in Fig. 4:

1. The composite component is stopped, through its lifecycle controller. This causes the stop of the inner component(s) as well, but the membrane is still working, and the incoming functional/non functional calls are correctly enqueued in the queque of the ProActive active object implementing the membrane. Note that, according to the ProActive implementation of composite

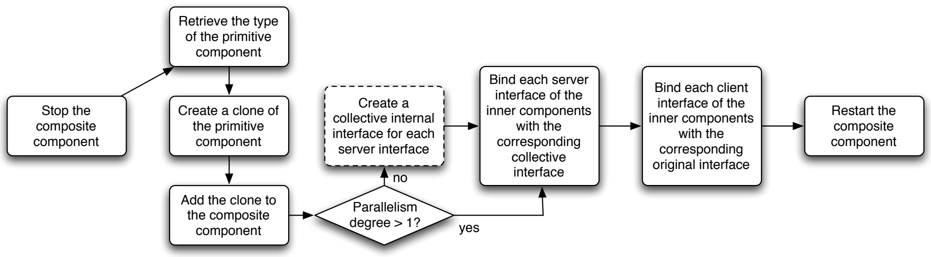
Stop the composite component
Retrieve the type of the primitive component
Create a clone of the primitive component
Add the clone to the composite component
Parallelism degree > 1?
no
yes
Create a collective internal interface for each server interface
Bind each server interface of the inner components with the corresponding collective interface
Bind each client interface of the inner components with the corresponding original interface
Restart the composite component

Figure 4: The sequence of actions implemented by ABC to create a new copy of the primitive component.

components, the incoming non functional calls are still served when the component is stopped.

2. The type of the primitive component is retrieved and a new instance is created and added to the composite one.

3. A new collective client interface is created for each functional server interface of the primitive component (which have a corresponding server interface in the composite component's membrane). This (internal) interface requires a particular management in the ProActive implementation of the membrane. It is possible to modify such behavior to manage the scheduling of the incoming calls. There is not an equivalent internal collective server interface at this point, because there are no particular implementation issues in connecting several client interfaces (from the primitive components) to a server interface (the original internal interface in the membrane). Moreover, we decided not to force the wrapping composite to have external multicast interfaces.

4. The new and old component interfaces are bound to the interfaces in the membrane of the composite component, and the whole component is restarted.

6.2 Autonomic Manager

A GCM component with active behavior must expose the component and some basic controllers (AC, BC, CC, LC) along with two additional interfaces:

```
interface AutonomicServerManager {
    any commitContract(String qosContract);
}

interface AutonomicClientManager {
    any raiseViolation(any violationId);
}
```

The `commitContract` method takes a QoS contract as input, and activates the steering of the behavior of a component to adapt itself to the requirements of the contract. If at runtime this contract is not respected, the `raiseViolation` method is used to signal this event.

The Autonomic Manager entity is responsible for the strategy to enforce at runtime a particular QoS contract. In doing so, it exploits the component's Autonomic Behavior Controller in order to inspect the non-functional status of the components and to trigger corrective actions to respect the contract. Logically, it is part of the membrane of a component because it is involved with the non-functional management, but it should have its own lifecycle, independent from the lifecycle of the component it is attached to.

An Autonomic Manager for the farm-like composite component discussed in the previous section has been implemented. The AM is implemented as a special inner component of the composite one, in such a way to:

- expose non-functional client interfaces (e.g. `AutonomicClientManager`) which can not be specified in the current ProActive implementation;
- exhibit a lifecycle independent from the lifecycles of the primitive components.

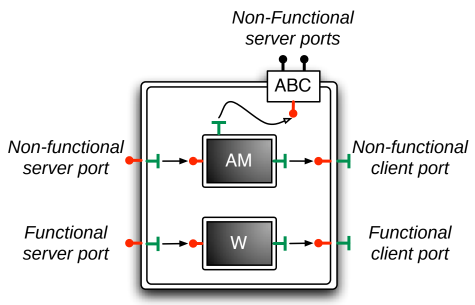
Non-Functional server ports
ABC
Non-functional server port
AM
Non-functional client port
Functional server port
W
Functional client port

Figure 5: The component in Fig. 3 with the Autonomic Manager.

The AM is connected to the ABC through an additional binding to an internal server interface on the farm component membrane. This binding is exploited to invoke the ABC specific operations. Moreover, all the composite components exhibiting an active autonomic behaviour, thus including an AM, have a customized lifecycle controller (LC). This is mainly due to the fact that the standard GCM-ProActive LC cannot selectively stop the inner components. The customization enable to stop all the inner component but the AM, which cannot be stopped because it is driving the component reconfiguration.

7 Preliminary Experiments

In this section we present preliminary experiments of the farm behavioural skeleton. These experiments aim to evaluate basic farm behaviour skeleton functionality and speedup. All the experiments have been conducted using a cluster with 9 nodes: one gateway/client/management node and eight processing nodes. Each node consists of two (hyper-threaded) 2Ghz Xeon CPUs and 1Gbyte DDR400 RAM on-board. The cluster uses for the internal communications Fast-Ethernet adapters (100 Mbit-s/s). All the cluster nodes run Fedora 5 Linux operating system. The kernel version installed during the test sessions was the `2.6.20-1.2316` with the SMP support. The ProActive jar version used was the 3.2 and the other jars used were the ones bundled with the ProActive distribution. All the experiments were performed using **`rmissh`** as ProActive communication protocol. The GCM code used in the experiments is described in [2]. The experiments were performed using several different computation to communication time ratio (i.e. computational grain). The results reported in Fig. 6 and Fig. 7 show that good speedups can be achieved provided the workers exhibit a large grain. Currently, this seems to be largely due to some inefficiency in the ProActive communication machinery (that relies on Java RMI and Java serialization mechanism). As evident from Fig. 8, we currently succeeded to saturate just a wee fraction of the theoretical network bandwidth. In this regard,

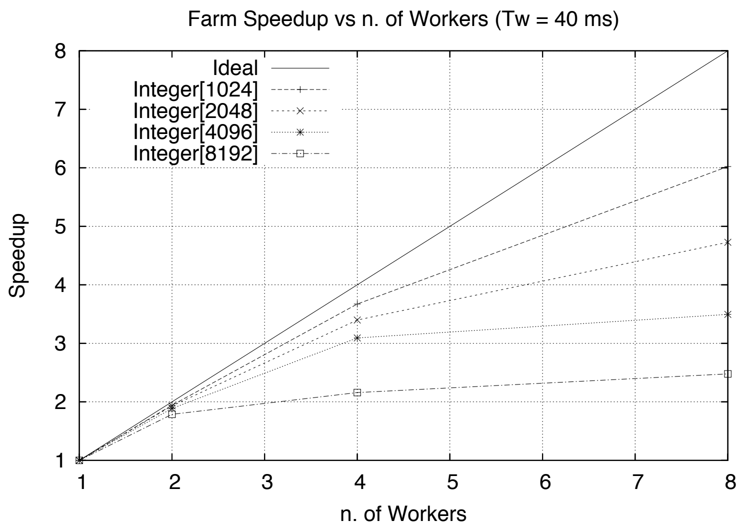

Figure 6: Speedup of the farm skeleton with respect to the number of workers.

the well-know Java serialization inefficiency is not likely to be the prominent cause of the poor communication throughput. As a matter of fact, the communication throughput figures shown Fig. 8 for native types (e.g. `int`) and full-fledged objects (e.g. `java.lang.Integer`) do not significantly differ, and both of them are well below expected values. We expect to further investigate the phenomena together with WP2 project partners. Also, we are currently planning a new set of experiments in order to evaluate behaviour skeleton reconfiguration overhead and re-activeness, i.e. the ability of behavioural skeletons to drive timely and low-latency parallelism degree variation to address non-functional requirements [7].

8 Conclusion

The challenge of autonomicity in the context of component-based development of grid software is substantial. Building into components autonomic capability typically impairs their reusability. We have proposed behavioural skeletons as a compromise: being skeletons they support reuse, while their parametrization allows the controlled adaptivity needed to achieve dynamic adjustment of QoS while preserving functionality. We have described how these concepts can be applied and implemented within the GCM. We have introduced a significant set of skeletons, together with their self-management strategies. We presented the GCM implementation of a class of those (functional replication class), that have been exemplified via the farm skeleton. The presented behavioural skeletons have been implemented in GCM-ProActive [10], in the framework of the WP3 of the GridCOMP project and are currently under extensive experimental evaluation. Preliminary results, confirm the feasibility of the approach.

References

[1] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, P. Kilpatrick, D. Laforenza, and N. Tonellotto. Behavioural skeletons for component autonomic management on grids. In *CoreGRID Workshop on Grid Programming Model, Grid and*

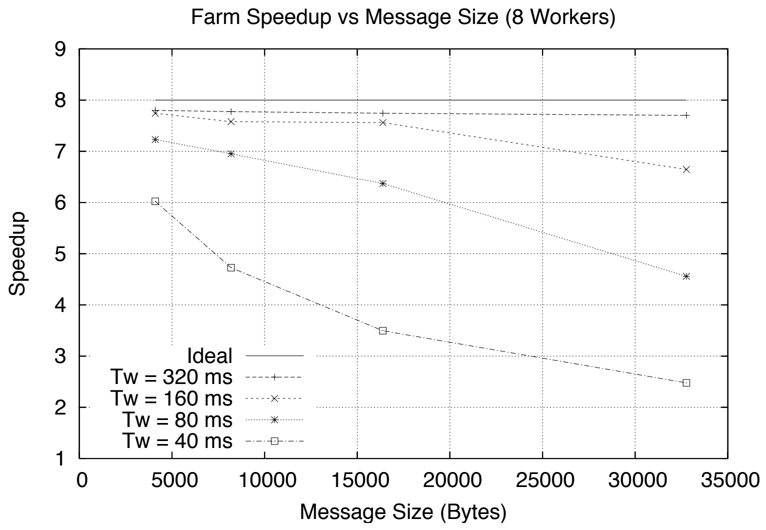

Figure 7: Speedup of the farm skeleton with respect to the size of messages.

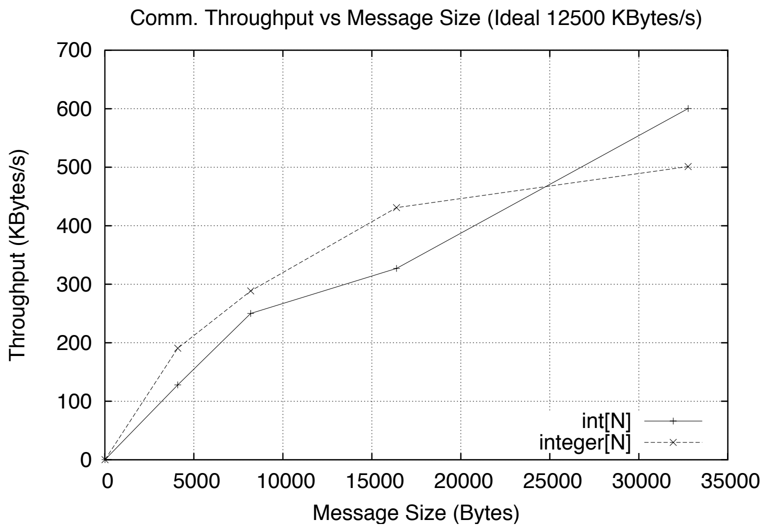

Figure 8: GCM-ProActive communication time evaluation on a simple producer-consumer schema for `int[]` and `Integer[]` data types.

P2P Systems Architecture, Grid Systems, Tools and Environments, Heraklion, Creete, Greece, 2007.

[2] M. Aldinucci, S. Campa, and P. Dazzi. *D.NFCF.02 – NFCF early prototype (with basic non functional features)*. GridCOMP STREP deliverable D.NFCF.02, June 2007.

[3] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*, chapter 10, pages 230–256. Springer, Jan. 2006.

[4] M. Aldinucci and M. Danelutto. Algorithmic skeletons meeting grids. *Parallel Computing*, 32(7):449–462, 2006. DOI:10.1016/j.parco.2006.04.001.

[5] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Management in distributed systems: a semi-fomal approach. In A.-M. Kermarrec, L. Bougé, and T. Priol, editors, *Proc. of 13th Intl. Euro-Par 2007 Parallel Processing*, LNCS, Rennes, France, Aug. 2007. Springer. To appear.

[6] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST grid-aware components. In B. D. Martino and S. Venticinque, editors, *Proc. of Intl. Euromicro PDP 2006: Parallel Distributed and network-based Processing*, pages 221–230, Montbéliard, France, Feb. 2006. IEEE.

[7] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In J. C. Cunha and P. D. Medeiros, editors, *Proc. of 11th Intl. Euro-Par 2005 Parallel Processing*, volume 3648 of *LNCS*, pages 771–781. Springer, Aug. 2005.

[8] F. André, J. Buisson, and J.-L. Pazat. Dynamic adaptation of parallel codes: toward self-adaptable components for the Grid. In *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series. Springer, Jan. 2005.

[9] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schütt. On adaptability in grid systems. In *Future Generation Grids*, CoreGRID series. Springer, Nov. 2005.

[10] F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for grid programming. In V. Getov and T. Kielmann, editors, *Proc. of the Intl. Workshop on Component Models and Systems for Grid Applications*, CoreGRID series, pages 97–108, Saint-Malo, France, Jan. 2005. Springer.

[11] J. Bosch. Superimposition: a component adaptation technique. *Information & Software Technology*, 41(5):257–273, 1999.

[12] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[13] CoreGRID NoE deliverable series, Institute on Programming Model. *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, Feb. 2007.

[14] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In W. Löwe and M. Südholt, editors, *Proc of the 5th Intl Symposium Software on Composition (SC 2006)*, volume 4089 of *LNCS*, pages 82–97, Vienna, Austria, Mar. 2006. Springer.

[15] S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGRID series. Springer, Nov. 2005.

[16] K. Kennedy, M. Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chien, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, and R. Wolski. Toward a framework for preparing and executing adaptive Grid programs. In *Proc. of NSF Next Generation Systems Program Workshop (IPDPS 2002)*, 2002.

[17] Next Generation GRIDs Expert Group. *NGG3, Future for European Grids: GRIDs and Service Oriented Knowledge Utilities. Vision and Research Directions 2010 and Beyond*, Jan. 2006.

[18] ObjectWeb Consortium. *The Fractal Component Model, Technical Specification*, 2003.

[19] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri. AutoMate: Enabling autonomic applications on the Grid. *Cluster Computing*, 9(2):161–174, 2006.

[20] S. M. Sadjadi and P. K. McKinley. Transparent self-optimization in existing CORBA applications. In *Proc. of the 1st Intl. Conference on Autonomic Computing (ICAC'04)*, pages 88–95, Washington, DC, USA, 2004. IEEE.

[21] E. Truyen, B. Jørgensen, W. Joosen, and P. Verbaeten. On interaction refinement in middleware. In J. Bosch, C. Szyperski, and W. Weck, editors, *Proc. of the 5th Intl. Workshop on Component-Oriented Programming*, pages 56–62, 2001.

[22] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.