UNIVERSITÀ DI PISA
DIPARTIMENTO DI INFORMATICA

TECHNICAL REPORT: TR-06-13

Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST Implementation

M. Aldinucci*, C. Bertolli†, S. Campa†, M. Coppola*, M. Vanneschi†, L. Veraldi†, C. Zoccolo†
*Institute of Information Science and Technology
CNR, Via Moruzzi 1, Pisa, Italy
Email: {marco.aldinucci, massimo.coppola}@isti.cnr.it
†Dep. of Computer Science, University of Pisa
Largo Bruno Pontecorvo 3, 56122, Pisa, Italy
Email: {bertolli,campa,vannesch,veraldi,zoccolo}@di.unipi.it

August 15, 2006

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST Implementation

M. Aldinucci*, C. Bertolli†, S. Campa†, M. Coppola*, M. Vanneschi†, L. Veraldi†, C. Zoccolo†
*Institute of Information Science and Technology
CNR, Via Moruzzi 1, Pisa, Italy
Email: {marco.aldinucci, massimo.coppola}@isti.cnr.it
†Dep. of Computer Science, University of Pisa
Largo Bruno Pontecorvo 3, 56122, Pisa, Italy
Email: {bertolli,campa,vannesch,veraldi,zoccolo}@di.unipi.it

***Abstract*— We present the concept of adaptive super-component as a hierarchy of components to which a well-known parallel behavior is associated. The proposal of a super-component feature is part of the experience we gained in the implementation of the ASSIST environment, which allows to develop self-configuring and optimising, component-based applications following a structured and hierarchical approach. We discuss how such approach to Grid programming influenced the design of the GCM component model.**

I. Introduction

Grids Computing platforms offer the option to run complex and multidisciplinary applications, exploiting aggregate software and hardware resources that are physically available at no single computation center. On the other hand, the Grid is a highly dynamic platform, where resources availability changes over time while the program is executing. This makes adaptivity an essential feature in order to achieve high performance and efficiently exploit the available resources.

Adaptivity [1] means that an application is (or its components are) able to change its configuration at run-time, preserving the semantics of the ongoing computation and dynamically adapting to a specific need.

The *autonomic computing paradigm* [2], [3] targets self-managing systems and applications, as defined by four aspects that should be implemented by each system component:

- Self-Configuring: a component is self-configuring if it is able to handle configuration goals in spite of the underlying platform heterogeneity.
- Self-Healing: A component is self-healing if it is able to provide its services in spite of failures of any kind.
- Self-Optimising: a component is self-optimising if it adapts its configuration and structure in order to achieve the best/required performance.
- Self-Protecting: A component is self-protecting if it is able to predict, prevent, detect and identify attacks, and to protect itself against them.

All these aspects can be studied as a whole in the framework of autonomic computing, where support code, and abstractions provided by the programming environment, react dynamically in order to obey multiple kinds of constraints.

The Grid Component Model (GCM) is a proposal for a component model oriented to Grid platforms, being developed within the framework of the CoreGRID Network of Excellence (NoE). Part of our contribution to the GCM component model is in the set of abstractions needed to express autonomic behaviour taking into account the aforementioned aspects in a common and consistent way. Our contribution is based on the experiences we have made in the development of the ASSIST parallel programming environment [4], [5] and of the component model developed in the *Grid.it* research project. In this paper we will introduce the notion of super-component. In ASSIST an adaptive super-component is the result of a hierarchy of structured components provided with an embedded parallel behavior. By instancing a super-component the programmer selects functionalities as well as a well-known parallel behavior (with given performance issues). Moreover, each super-component represents a well-known pattern of parallelism whose composite components can be automatically managed in their non-functional aspects, without the programmer intervention. Last but not least, since a super-component exposes also adaptive features, it is able to manage itself in order to follow self-configuration, self-optimization, self-healing and self-protecting targets.

ASSIST shares with GCM many features w.r.t. autonomic behaviour. On the one hand, GCM is currently a proposal for a framework fully supporting hierarchical autonomic components, in order to realize component-based autonomic Grid applications. Grid.it components, on the other hand, already provide self-optimising and self-configuring behaviour through a hierarchy of user-configurable manager modules, an approach that already enables building HPC Grid applications.

In Sec. II we discuss previous work related to component models and (self-)adaptive behaviour in Grid programming also related to super-component abstractions. Starting from the autonomic paradigm, which is the matter of Sec. III,we give a first definition of the kind of interfaces that an autonomic

component frameworks should provide, in order to support hierarchical composition and all aspects of autonomicity.

In Sec. IV we describe the architecture and the implementation of the ASSIST [5] programming environment, which provides self-configuring and self-optimising parallel software components. In this section, we will focus on the role of the hierarchical organization of component managers as well as on the notion of super-components. In Sec. VI we relate the ASSIST approach to the ongoing definition of the GCM component model and we will underlining their common aspects. Moreover, ASSIST supports the development of interoperable applications onto heterogeneous platforms. In Sec. V we show with test results the self-management features of ASSIST programs at run-time. Reconfiguration steps are fully transparent to the application programmer, and may be automatically triggered by ASSIST run-time support according to a user defined strategy [6]. Sec. VII concludes our presentation and outlines future work directions.

II. Related Work

High-level programming environments for grid aim at moving most of the grid specific efforts needed while developing high-performance grid applications from programmers to grid tools and run time systems. A fundational proposal is represented by the CORBA Component Model (CCM) [7], followed by the Condor [8] experience from which we were initially inspired in the design of the ASSIST component [4]. GridCCM[9] is an extension of CCM supporting parallel components with distributed data and communication optimizations differing from our approach because of our focus on adaptivity issues. In this sense and with respect to dynamic reconfiguration and reoptimization, we share common goals with the GrADs project [10], besides our programming model proposes a more severe component structure. Common Component Besides a clear component model, explicit component composition and run-time adaptivity, we also propose the notion of *super-components*, i.e. hierarchically structured components able to drive to application reconfigurability. ProActive [11], a Java implementation of the Fractal component model exploiting RMI calls as primary communication mechanisms, is a project that shares with us the super-component and adaptivity targets but, currently they do not offer library mechanisms (all works arounds RMI mechanisms). We mention Ibis [12] as another Java based programming environment offering a kind of *divide-and-conquer* super-component notion but not exploiting autonomicity issues. A similar approach is followed by the Higher-Order Components (HOCs), that are [13] are components offering the notion of components parameterized with data and code but they do not support autonomicity. Thus, a component expresses a behavioral schema that can be instantiated on the target architecture at hand by providing the corresponding code units. As seen below, the idea of having kind of higher-order components is also exploited in the Grid.it component model but we propose a higher level of abstraction where hierarchy takes an important role.

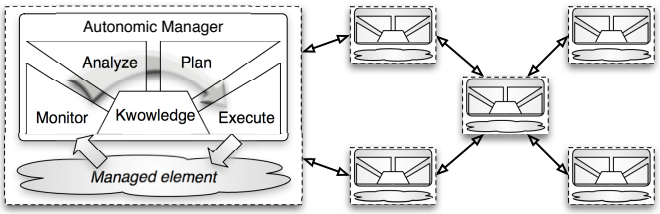

Fig. 1. Structure of an autonomic element. Elements interact with other elements and with human programmers via their autonomic managers [15].

III. Toward and autonomic Grid Component Model

As shown in Fig. 1, an autonomic element will typically consist of one or more managed elements coupled with a single autonomic manager that controls them. The managed element could be a hardware resource (storage, CPU, etc.), or a software resource, such as a Web service, or a software *component*. Control loops, which are known in optimization theory since (at least) the mid of the last century, can be used to apply component self managing. They split the optimization process into 1) a *monitoring* phase, where the symptoms are collected; 2) an *analysis* phase, where the current status is checked against the goal status; 3) a *planning* phase, where a plan is created to enact the desired alterations according to some policies; and 4) the *execution* phase, which provides the mechanisms to schedule and perform the necessary changes to the system [14], [15]. Truly autonomic systems are years away, although in the nearer term, autonomic functionality will appear in software, especially in very complex systems as Grid-aware applications. In particular, early autonomic system may threat self-optimization, self-healing, self-configuration and self-protection as distinct aspects, with different solutions that address each one separately.

The design of the component specification is driven by several factors. First of all, we want to allow the reuse in parallel components of existing code. Thus the component definition must be abstract w.r.t. the component implementation and to the run-time support of its implementation language. Equally important, we want to describe autonomic behaviour, so that components can be orchestrated in a consistent way in order to pursuit Quality of Service goals for the entire application. It should be possible to exploit both adaptation mechanisms coded by hand by experienced programmers, and standard ones provided by high-level programming languages.

To face this two-fold problem, the component model defines *manager* entities, and a proper set of *non-functional interfaces*. The manager of a component implements its non-functional ports, and it is supposed to provide the component with some autonomic features. Non-functional ports standardize the interactions between different layers of application management.

Each quality aspect of an application clearly depends on the quality level provided by every component, but the relation can be complex, so that local choices taken in different components, using only local information, hardly can lead to the optimum. A straightforward solution, anticipated in

Sec. III, is to leverage the component hierarchy of the application to coordinate its autonomic behaviour. To fulfill an autonomic goal, a component should generally rely on its subcomponents, and generate sub-goals that will be delegated to them.

Applying autonomicity hierarchically is particularly important for a hierarchical and distributed component model like GCM or Grid.it are. The kind of hierarchy and the management structures we can build depend on the *abstraction level* provided by the autonomic interfaces that a component frameworks defines.

1) The non-functional ports may expose/provide low-level information/commands to external world to steer the component behaviour. As an example, component sensor values can be exposed to an external manager (or a user interface) which can tune parallelism and replication degrees, mapping, fail-stop tolerance level for a component. This approach places all the burden of managing the component behaviour on the external controller, but it also allows to implement autonomic behaviour for non-autonomic components, or to override the component's autonomic mechanism. This choice, if taken to extremes, results in a system where autonomic control is actually centralized, and monitoring information has to be propagated, deceiving the encapsulation property of components.
2) Alternatively, non-functional ports may expose/provide interfaces toward the autonomic manager of each component, as a mean to assign goals to pursue, described by formal, high-level contracts, and to collect information needed for non-local strategies. As an example, a component may accept a QoS contract, and try to obey it autonomously, possibly triggering events whenever the contract can no longer be fulfilled. In this case the control of component configuration is indirect. Proper filtering of events and coordination of these high-level interfaces is needed in order to enable proper subgoal delegation without breaking component encapsulation.

As we will see in the following, ASSIST components and super-components may be equipped with a *manager*, while GCM hierarchical components can have *component controllers*. In both models, these managing entities can exploit sub-component's monitoring information (introspection) and autonomic capabilities, at the high and low abstraction levels, in order to achieve global QoS, by assigning contracts as goals to autonomically pursue, as well as to steer the adaptation mechanism of a component when non-local strategies have to be employed to manage less autonomic (legacy) components.

In the following sections we will describe how such aspects are modeled in the ASSIST environment by exploiting *adaptive supercomponents* as the result of a hierarchy of structured *managers* and how this contribution will be mapped onto the definition of the GCM model.

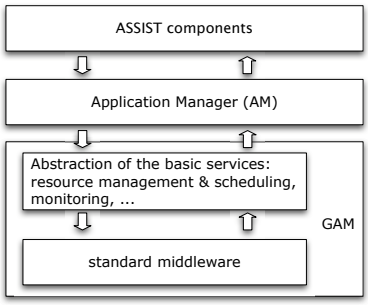

Fig. 2. ASSIST software architecture.

IV. The ASSIST framework

ASSIST currently supports the *Grid.it* component model (developed within the *Grid.it* Italian national project) which shares several features with the forthcoming GCM. A complete porting of the ASSIST to meet the GCM is planned in the near future. In the rest of the section we sketch the ASSIST programming environment, starting from its parallel coordination language, its modular architecture and the support for components, to introduce adaptive supercomponents driven by an ASSIST hierarchy of managers and end up with the description of ASSIST self-managed QoS for performance and self-configuration.

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data.

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that run as *Virtual Processes* (VPs) activated by items arriving from the input streams. VPs may synchronize implicitly by activation, or through explicit barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran).

A *parmod* may behave in a data-parallel (e.g. SPMD/forall/apply-to-all) or task-parallel way (e.g. task farm), and it may exploit a distributed shared state, which survives to VPs lifespan. More details on the ASSIST coordination language can be found in [4], [5].

The ASSIST compiler translates an arbitrary graph of ASSIST modules into a network of processes. Sequential modules become sequential processes, while parallel modules are turned into a parametric networks of processes, each one behaving as the execution engine for the VPs of a given parmod. Besides, processes devoted to manage module adaptivity and QoS control are added to the network (see [6], [16] for any further details).

As we shall see from the next sections, ASSIST modules exhibit the same kind of non-functional interfaces of GCM components, and once wrapped within Grid.it components they can be dynamically wired one another.

A. The Grid.it Component: Modules, Non-Functional Interfaces and Managers

A single parmod or an arbitrary ASSIST graph of modules can be declared as *Grid.it* component. A Grid.it component is characterized by *provide* and *use* ports, as well as by *non-functional ports*, which are related to component QoS control. Each port of an ASSIST component may be configured to behave as endpoint of one-way stream connection, RPC method, or event channel. A Grid.it component may also interoperate with Corba/CCM (via IIOP based RPC) components or Web services (via HTTP/SOAP).

The software architecture of the ASSIST component-based parallel programming environment is organized as shown in Fig. 2. The run-time environment of ASSIST 1.3 is implemented on top of a *Grid Abstract Machine* (GAM). The GAM implements *abstract services*, i.e. the functionalities needed by the programming environment to support high-performance, component-based Grid-aware applications. These regard resources discovery, management and monitoring; components deployment, run, and wiring; routing of communications through networks with private addresses.

Whether possible, these services rely on the underlying Grid middleware, which are just abstracted out at the GAM level. In other cases, GAM services *extend* Grid middleware services (e.g. monitoring) [17].

Grid.it components natively implement several non-functional ports, for both introspection and autonomic purposes. The non-functional interfaces publicly expose either an RPC or an event-based semantics, involving subscriptions and following gather of required information. The standard RPC-based interfaces for low-level dialog with Grid.it components and super-components are:

- request for monitoring measurements;
- describe the current parallel layout of the component;
- apply a user-provided reconfiguration script;
- suspend/resume the component computation;
- stop the computation, releasing all involved grid resources allocated for the component.

Any Grid.it component can directly be asked to report its instantaneous performances or details about its own modules, their location on the grid and current parallel behaviour. Aside the autonomic behavior of the manager hierarchy, users may directly interfere with the decision process for reconfigurations and impose a sequence of changes for the component to obey to: e.g., explicit addition of parallelism up to a certain degree for a farm application manager.

Components can explicitly be suspended (and subsequently resumed or stopped), in a correct manner w.r.t. the semantics of the parallel computation. This functionality is exploited in farm application manager to allow a high-performance implementation of autonomic behavior. In fact, farm managers allow worker components to be removed as well as added dynamically. To optimize performance, considering the non negligible overhead of component package transmission and loading on remote machines, the implemented policy for dynamicity in farm-like application managers currently choices to suspend, rather than barely stop and unload, exceeding workers, up to a certain limit.

In all the frequent scenarios where component input pressure varies during executions, and cuts in the parallelism degree are likely to be followed by new increments, this approach may lead to much better results, since resuming a suspended component is orders of magnitude faster than instantiating a new one.

Finally, users may like to stop a component running on a certain grid site, in order to execute an identical copy of it elsewhere, where performance/cost rate may be better. Gracefully stopping a component is a low-level mechanism to implement stateless migration.

The event-based interfaces, instead, provide for:

- subscription for continuous performance monitoring measures, at regular time intervals:
- subscription for QoS contract violations.

These introspection interfaces allow either a user interface, or a component autonomic runtime support on users' behalf, to monitor in real-time the execution of controlled components.

The most interesting non-functional port for Grid.it components is the RPC one allowing for users (or managers, as well) to dynamically change the hierarchical QoS contract, for the application as a whole or for portions of it.

B. The hierarchy of Managers

As shown, component nesting provides the application with a logically unitary managing infrastructure, actually composed of the set of managers along the hierarchy of components, which operate in distributed and cooperative fashion.

We call CAM (Component Autonomic Manager) the manager of a component or super-component. As mentioned in the previous sections, ASSIST modules exhibits the same non-functional interface of Grid.it components, and can be considered as part of the managing infrastructure. We call MAM (Module Autonomic Manager) the manager of a module. We call Application Manager (AM) the CAM at the root of the manager hierarchy, since it (indirectly) controls the whole application, and coordinates the QoS of the whole application through CAMs and MAMs. The AM also assumes the role of the interface toward the user. Clearly, nothing prevents the AM itself to be composed of a set of distributed entities cooperating to achieve the same goal of enforcing a given QoS for the whole application [16].

Fig. 3 shows an application in which we recognize four components, component a consisting of modules M_1, component b consisting of modules M_2, and component c consisting of modules M_3 and M_4. The whole application exposes a provided port. Each module and component has an associated management entity, respectively a CAM or a MAM, arranged as a tree having the AM as its root.

Each CAM applies control strategies at the level of the associated component, leveraging on non-functional ports of the nested components. Whenever nested components do not

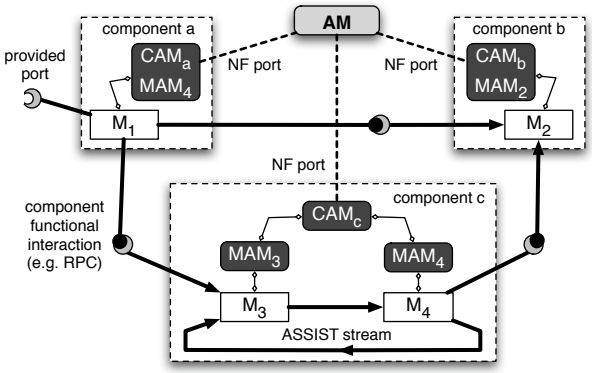

Fig. 3. Four interacting Grid.it components.

exploit any significant mechanism for adaptation and reconfiguration, the CAM can possibly implement strategies based on dynamic component creation and wiring functionalitiess provideded by the component model. As a concrete example, a component wrapping a legacy MPI program will likely miss adaptation functionalities (its non-functional ports will be no-ops). If the legacy component state can be saved or disregarded, an outer CAM may create on the fly a new instance of the component, with a different configuration and mapping, and substitute the new version to the old one. A CAM can also receive proposals of restructuring by the child CAMs *(monitor)*. In this case, the CAM has to apply a global performance model in order to detect the need to restructure more children modules and devise a good solution *(analyze & plan)*. Recursively, a CAM can receive reconfiguration requests from father CAMs, and can send them reconfiguration proposals *(execute)*. The root manager (AM) is the eventual responsible for the final decisions in the global reconfiguration control which, as seen, is a sort of parallel and asynchronous Divide&Conquer strategy applied along the hierarchical Application Manager structure.

The definition of a sound set of behavioral and interaction rules (that, once embedded in CAMs, will induce the desired global behavior) is under investigation. For instance, a general strategy enabling to make sound decisions at the lowest possible level in the CAM hierarchy may significantly improve management overhead. To this end, the analysis of the application graph in terms of a queuing network seems a promising approach [18], [19]. This approach enables the detection of bottlenecks in application DAGs or sub-DAGs exploiting a data-flow behavior, i.e. ASSIST components mainly interacting via streams.

C. Super-components

The advantages offered by a hierarchical structure of a component application based on the managers interactions, suggests a further abstraction step leading to the notion of super-component, i.e. a container that can host both other components or super-components. Grid.it super-components may be considered as higher-order or parametric components which can be instantiated at launch time with other components. They describe common computation paradigms (skeletons, actually). We have to point out that such common computation paradigms could be also described by manually composing Grid.it components into a graph a by caring about the encoding of the related managers. However, this is a complex and error prone programming phase. Super-component allow to provide the programmer with a well-known hierarchical structure of components whose manager can be automatically generated.

In other words, Grid.it super-components differ from other implementations of higher-order components (e.g. HOCs [13]) because they are transparent to the programmer, who is not required to care about internal behavior (such as scheduling and data distribution) since it is defined by the super-component kind (as usual for skeletons). As a consequence, it is also possible to equip the super-component with an automatically generated autonomic manager.

In the Grid.it model two kinds of super-components are currently defined:

a) DAG: it enables the wiring of components/super-components as nodes of a Direct Acyclic Graph, as a generalisation of the a pipeline parallel pattern.

b) Farm: it enables the replication of a given host component/super-component, and is functionally equivalent to the replicated component, exposing the same provided/used functional ports of the host. The farm can of course expose different non-functional ports. Each data item (call/pure data) on the farm provided functional ports is routed to the provided ports of one of the internal replicas. Data items from the replica's used ports are routed on the super-component ones. The replication degree can be changed at run-time via the farm non-functional ports, causing replicas to be spawn/deactivated, exploiting mechamisms and policies analogous to those used in parallel modules, described in this section and in Sec. V.

Super-components can be used with both event/one-way and RPC-style ports. However, they are particularly suited to be connected via one-way streams, describing a flow of data that is computed along different logical phases. In these case, wiring among components may be done via buffered ports (implemented via distributed queues), enabling multi-site deployment without a strict co-allocation mechanism.

Super-components turn the Grid.it component model into a hierarchical component model, according to GCM specification.

D. QoS contracts and its autonomic management

A Grid.it component may accept (statically or dynamically) a QoS contract via its non-functional ports. Currently, QoS contracts are described by a specific XML file, and include the specification of the processing bandwidth (service time) in stream-based computations, and/or the completion time, which is often more significant for non-stream computations. Such contract may be subject to constraints on the amount and on the kind of computing resources.

A Grid.it QoS contract carries a component QoS goal and the description on how it should be achieved. In particular:

- *Performance features:* a set of variables, which can be evaluated from module static information, run-time data, collected through monitoring, and performance model evaluation.
- *Performance model:* a set of relations among *performance features* variables, some of them representing the performance goal.
- *Performance goal:* a set of inequalities involving *performance features*.
- *Deployment annotations* describing processes resource needs, such as required hardware (platform kind, memory and disk size, network configuration, etc.), required software (O.S., libraries, local services, etc.), and other all strictly required constraints to enforce code correctness.
- *Adaptation policy:* a reference to the desired adaptation policy chosen among the ones available for the module. Standard adaptation policies are represented as algorithms and embedded within MAM code at compile time.

Among all possible QoS goals, Grid.it managers currently support the performance related ones that are achievable through adaptation within each parallel module. Aspects regarding modules coordination, as well as other QoS measures such as reliability, availability, and security are currently under investigation.

The performance models used in the ASSIST framework range from very simple and approximate analytical models, such as the one used to manage task farm parmods, to more complex models derived using advanced mathematical techniques, such as those derived in [6], [18].

As an example, the following is the QoS contract of the experiment in Fig. 4. For more details about Grid.it QoS contracts we refer back to [16].

Perf. features	QL_i (input queue level), QL_o (input queue level), T_{ISM} (ISM service time), T_{OSM} (OSM service time), N_w (number of VPMs), $T_w[i]$ (VPM$_i$ avg. service time), T_p (parmod avg. service time)
Perf. model	$T_p = \max\{T_{ISM}, \sum_{i=1}^{n} T_w[i]/n, T_{OSM}\}$
Goal	$T_p < K$
Deployment	arch = (i686-pc-linux-gnu $\vee$ powerpc-apple-darwin*)
Adapt. policy	goal_based

Since an ASSIST super-components are pre-defined hierarchical structures of components, the autonomic management discussed can be applied also to such structures, thus allowing the ASSIST framework to expose autonomic super-components.

V. ASSIST Implementation Results

In this section we will focus on adaptation for ASSIST parallel modules with reference to the satisfaction of user-provided QoS constraints. We will show quantitative results and discuss related experiences in the literature.

A. ASSIST native adaptivity

As reported in [20], the ASSIST support natively provides for a wide range of dynamic adaptations for parallel

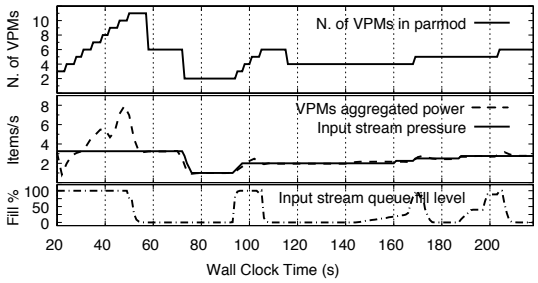

Fig. 4. Experiments on ASSIST adaptivity: Farm reconfigurations guided by QoS contract changes

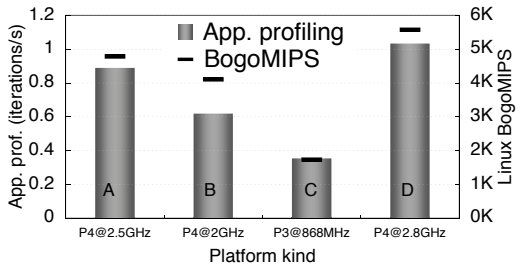

Fig. 5. Experiments on ASSIST adaptivity: Performance of chosen machines on a non-dedicated execution environment

modules: new processes belonging to the VPM class can be added/removed at runtime within a parmod, and can be migrated across heterogeneous platforms, using a specifically optimized checkpointing strategy. This allows to exploit remapping strategies to balance the workload in data-parallel computations

No matter when a change request is issued by user or runtime support, the involved module is actually able to reconfigure itself only in special conditions. In such time windows, which are called *reconf-safe* points, the parmod state is consistent, i.e. it is completely defined by the value of its attributes, and no communications involving data are pending.

Notably, the runtime does not introduce any additional synchronization, apart from those required by program semantics. It rather delays reconfiguration execution until the next natural reconf-safe point is reached.

Reconfiguration actions are implemented and optimized for each parmod taking into account its parallel computation semantics. Migration overhead is especially dependent on the knowledge we can infer from the high-level specification of the computation that a parmod provides.

B. Quantitative results

To evaluate the effectiveness of our approach we report experiences about a farm and a data-parallel parmod (Fig. 4–5). The former farms out a dummy sequential function with 2s average service time; the latter computes an iterative (`forall`) reduction on internal shared state.

Farm parmod is executed with an initial, relatively strict QoS constraint over the overall module service time (Fig. 4). The QoS contract for the parmod is changed twice by

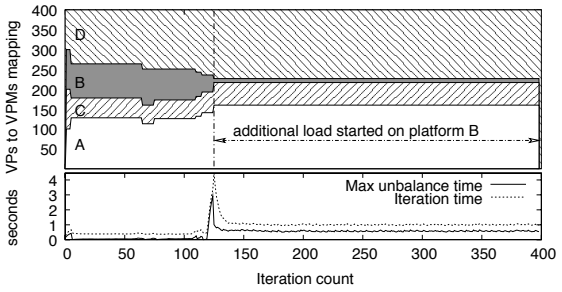

Fig. 6. Experiments on ASSIST adaptivity: Data-parallel rebalancing

user. The first time, about 70s after computation start, a more relaxed contract is submitted. The runtime support consequently mandates a reduction of the parallelism degree to free the exceeding resources. The second time, QoS gets tighter, thus fresh resources are recruited and new VPM processes launched on them, in order to meet the user requirements.

Data-parallel computation is distributed on four heterogeneous machines in a non-dedicated execution environment (Fig. 5 shows their relative performance). When a PE is artificially overloaded (Fig. 6, ray-tracing and code compiling starts on node B) the autonomic runtime redistributes the workload trying to re-balance the computation. The support decides to shrink the partition of shared data assigned to the overloaded PE for the rest of the execution, distributing the exceeding workload proportionally to the performance of the other PEs.

These experiments show that the approach is feasible for data-parallel computations, and that good results are obtained for the task-parallel ones, for which we found effective adaptation policies.

C. Related experiences

In order to find earliest examples of dynamically reconfigurable code we shall go back until the eighties; proposals included process migration environment both at the OS level ([21]) with kernel modules, and at the application one, with user libraries providing checkpointing API ([22], [23]). Main checkpointing drawback is that it's not generically suitable for Grid and high-performance, distributed executing environments. More recently, Java bytecode portability has been exploited to equip programs with load balancing facilities ([12]) or provide user-level mechanisms for process migration ([24]). The most interesting and promising approach to dynamicity in Grids appeared in [25], where the GrADS project is presented. The runtime facilities include contract negotiators and configuration optimizer. Our approach is quite different in that we provide for a transparent dynamicity (no specific programming efforts required), with sensibly better performance (we do not stop the complete application), and designed and implemented an autonomic runtime system, able to cope with user-defined QoS constraints.

```
<Binding>
 <Components>
  <Component refId="InputComponent">
     <LaunchRef ref="inputPkg"/>
  </Component>
  <Component refId="DataParallelComp">
     <LaunchRef ref="dataParallelPkg"
                dataAAR="calibration"/>
  </Component>
 </Components>

 <Connectors>
  <Stream refId="Channel">
     <BasicType>long</BasicType>
  </Stream>
 </Connectors>

 <Dependencies>
  <Component refId="InputComponent">
     <Produce>
       <Stream refId="Channel" origName="out"/>
     </Produce>
  </Component>
  <Component refId="DataParallelComp">
     <Consume>
       <Stream refId="Channel" origName="src"/>
     </Consume>
  </Component>
  </Dependencies>
</Binding>
```

Fig. 7. A typical composition in a pipeline application

D. Component-based applications

Applications are expressed at the highest level, as simple interconnection of black-box components as well as hierarchical compositions of managers, where users only have to define the correct bindings of declared functional interfaces. The functional behavior of parallel components may be easily and effortless described, naturally exploiting the ASSIST coordination language. The framework supports stream, event and RPC communication paradigms, and consequently provides for component interconnection at run-time by means of declared interface binding. The experiences achieved with the work on ASSIST has naturally driven us to focus more on to the *stream* nature of logical interconnection. Study about the other kinds of support is currently ongoing.

As an example of component composition, Fig. 7 shows a typical layout of a pipeline application, where two stages are defined and a logical connector is created, of type stream. In the simple formalism used to bind components' interfaces together no knowledge of internal component behavior, nor implementation, is actually required to produce the final packaging of the application. The configuration file exposed can be used as input for a pipeline Application Manager to be executed on Grid machines. Once run, users can interact with the top-level manager to alter the parallel behavior of the stages or monitor its performance in real-time.

VI. The GCM approach

The Grid Component Model (GCM in short) has been recently introduced by the Work Package 3 of the CoreGRID NoE, as a unified software component model for Computational Grids, currently at the level of a proposal.

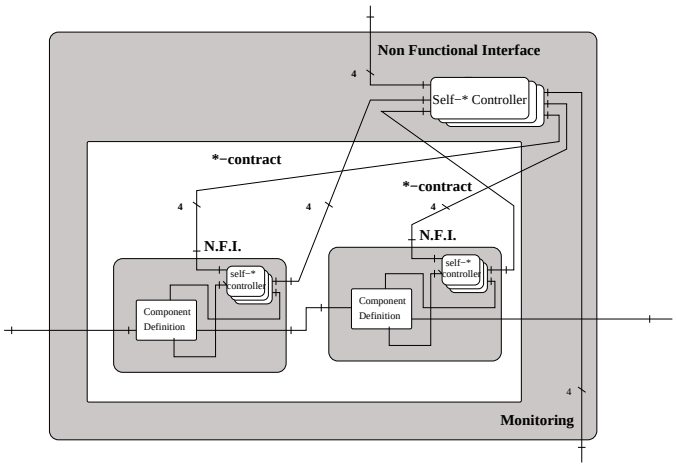

Fig. 8. The hierarchy of controllers inside a GCM composite component.

GCM is based on the Fractal component model [26], which is hierarchical. Components can be contained within other components, and the support of a component (its *membrane* in the Fractal terminology) can also be made up of other components. *Component controllers* are primitive Fractal components devoted to controlling their containing component, by means of internal non-functional interfaces which allow them e.g. to alter their host component configuration, and to intercept the communication flow between the inside and the outside of the host component.

To enhance component interoperability with legacy code, Fractal considers different levels of compliance of actual components with the implementation framework. GCM applies a similar approach also to autonomic behaviour, with the highest level of compliance being that of fully autonomic components, which implement the whole set of autonomic interfaces. At lower levels of compliance, components can implement a subset of the interfaces needed for autonomic behavior (e.g. steering or introspection interfaces).

The autonomicity of a GCM component can be characterized w.r.t. two main points:

1) non functional ports implements the interface through which the user can express, at an high-level, the desired behaviours/properties of the component. Non functional ports are implemented by a hierarchy of linked *component controllers*, whose structure reflects the component hierarchical structure (see Fig.8).
2) Each controller is related to a specific aspect of the autonomicity of the component. Controllers are implemented as sub-component of the whole component itself. In order to allow flexible dynamic replacing of support code within a component, *Dynamic* component controllers have been proposed which can be stopped and replaced at run-time, encapsulating specific policies or low-level functionalities. The specific implementation of controllers is thus configurable, and defines the autonomic behaviour of the component.

A. Autonomic Component Controllers

The GCM proposal defines a set of separate controllers that hierarchically implement the run-time support of the different autonomicity aspects, exposing interfaces through which contracts can be specified. The organisation of controllers is depicted in Fig. 8:

- The composite component provides a set of non functional interfaces bound to the controllers of the component itself.
- The controllers of the composite are bound to the controllers of the sub-components by means of subcomponent non-functional interfaces.
- Each sub-component controller directly monitors the subcomponent itself, and, when requested, provides this information to the composite component controller.
- Each sub-component controller directly manages the subcomponent.

As it can be seen, there is a quite clear similarity between GCM controllers and ASSIST manager, as well as between their hierarchical organization. However, GCM require a separate controller for each autonomicity aspect, while in ASSIST more attention is paid on the interactions between managers that coordinate them-self to reach a common autonomicity goal.

Another common aspect regards the role of non-functional interfaces. Non functional ports in the GCM proposal are each one related to a specific autonomicity aspect. Examples of the kind of requests acceptable by those interfaces are:

- a new level of performance of the component constraining the service time under a specified threshold, explicited by a new performance contract (what we call *self-optimization* in ASSIST)
- a requirement on the fault tolerance level provided, or on the fault recovery mechanism. That a contract can require a 99.9% probability of fail-free execution or mandate that a failure should cost less that one hour of computation (*self-healing* in ASSIST).
- a desired level of security for a protocol (e.g. the hardness of a cryptographic code) or a specific constraint on the kind of protocols/resources used *self-protection* (provided by ASSIST, too).
- the interface can accept the description of a goal, a parameter, or a procedure to adopt in order to configure the component (*self-configuring* in ASSIST).

The GCM framework proposal does not define how the different contracts can interact, even though it is clear that they do in the general case: for instance, a performance contract can be satisfied if resources are available (thus we may need to self-configure new ones) and as long as the overhead due to fault tolerance is not too high.

It is thus a matter of research to understand how contracts specified at the top of the hierarchy are translated into goals for the leaf components, and how to prevent or control interactions between different controllers, especially if we take into account that dynamic component controller can be replaced at

any time. This is a quite powerful feature but it also adds to the complexity of the problem.

As previously mentioned, GCM also supports deriving full autonomic behaviour from non-autonomic components. Controllers can choose to expose outside of the component the steering interfaces that are usually available only from inside, e.g. allowing to modify the parallelism degree of a subcomponent. Devolving control to an outer entity can break autonomicity, but also allows us to develop an overall autonomic composition out of non-autonomic components (having a very limited controller support) by adding external *manager* components which play the autonomic controller role. This approach is just the same that has been pursued in the design of ASSIST and the *Grid.it* component model.

VII. Conclusions and Future Work

In this paper we have presented a general approach to autonomic grid components, which is common to the GCM and the *Grid.it* component models. We have described the ASSIST architecture with its managing hierarchy ensuring specific QoS aspects and we have introduced the concept of super-component as the result of a hirarchy of manager component embedding a given parallel behavior.

From the test results it is clear that ASSIST partially implements the features required in the GCM model, allowing structured design and deployment of component based applications over grids, with high performance and ensuring autonomic control w.r.t. performance and optimization.

There are however differences and open issues which still have to be investigated. GCM models autonomicity thanks to separate a controller for each aspect: it is not yet clear how the different hierarchies related to the aspects will interact.

Even in the simpler design adopted in ASSIST/Grid.it, where managers are not separate for the different aspects, it a critical issue to develop interaction schemes and techniques to break autonomic goals into cooperating and non-conflicting subgoals to be applied to lower levels of a component hierarchy.

Currently, GCM doens't adopt the concept of super-component but we are working on integrating it into the model.

Another issue is that even if the ASSIST approach to autonomic performance management is general, we still are more geared toward stream asynchronous communication. While this communication paradigm is quite efficient to exploit over grids, more general models and techniques have to be applied in order to tackle more general application structures.

At the moment we are working to enlarge the set of super-components, and experimenting with different policies and coordination protocols for the manager hierarchy. At the same time, since our experiences will contribute to the analysis and design of the GCM component model, we have planned to enhance the compatibility between the ASSIST/Grid.it environment and GCM, implementing a larger subset of it within our programming environment.

Acknowledgments

This work has been partially supported by Italian national FIRB project no. RBNE01KNFP Grid.it, by Italian national strategic projects *legge 449/97* No. 02.00470.ST97 and 02.00640.ST97, and byt the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

[1] M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo, "Parallel program/component adaptivity management," 2005, PARCO 2005, Malaga, Spain, to appear.

[2] A. Ganek and T. Corbi, "The dawning of the autonomic computing era," *IBM Systems Journal - Autonomic Computing*, vol. 42, no. 1, pp. 5–18, 2003.

[3] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart, "An architectural approach to autonomic computing," in *Proceedings of the International Conference on Autonomic Comput ing*, May 2004, pp. 2–9.

[4] M. Vanneschi, "The programming model of ASSIST, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, Dec. 2002.

[5] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo, "ASSIST as a research framework for high-performance Grid programming environments," in *Grid Computing: Software environments and Tools*, J. C. Cunha and O. F. Rana, Eds. Springer, 2006, pp. 1–32, iSBN 1-85233-998-5. Draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, 2004.

[6] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo, "Dynamic reconfiguration of grid-aware applications in ASSIST," in *11th Intl Euro-Par: Parallel and Distributed Computing*, ser. LNCS, J. C. Cunha and P. D. Medeiros, Eds., vol. 3648. Lisboa, Portugal: Springer Verlag, Aug. 2005, pp. 771–781.

[7] T. C. . C. home page, `http://ditec.um.es/~dsevilla/ccm/` .

[8] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., December 2002.

[9] A. Denis, C. Pérez, T. Priol, and A. Ribes, "Bringing high performance to the corba component model," in *SIAM Conference on Parallel Processing for Scientific Computing*, February 2004.

[10] S. Vadhiyar and J. Dongarra, "Self adaptability in grid computing," *Concurrency & Computation: Practice & Experience*, vol. 17, no. 2–4, pp. 235–257, 2005.

[11] F. Baude, D. Caromel, and M. Morel, "On hierarchical, parallel and distributed components for Grid programming," in *Workshop on component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds., ICS '04, Saint-Malo, France, June 2005.

[12] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: a flexible and efficient java-based grid programming environment," *Concurrency & Computation: Practice & Experience*, vol. 17, pp. 1079–1107, 2005. [Online]. Available: http://www.cs.vu.nl/~kielmann/pubs.html

[13] S. Gorlatch and J. Dünnweber, "From grid middleware to grid applications: Bridging the gap with HOCs," in *Future Generation Grids*, ser. CoreGRID series, V. Getov, D. Laforenza, and A. Reinefeld, Eds. Springer-Verlag, Nov. 2005.

[14] A. Andrzejak, A. Reinefeld, F. Schintke, and T. Schütt, "On adaptability in grid systems," in *Future Generation Grids*, ser. CoreGRID series, V. Getov, D. Laforenza, and A. Reinefeld, Eds. Springer-Verlag, Nov. 2005.

[15] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[16] M. Aldinucci, M. Danelutto, and M. Vanneschi, "Autonomic QoS in ASSIST grid-aware components," in *Proc. of Intl. PDP 2006: Parallel Distributed and network-based Processing*, Euromicro. Montbèliard, France: IEEE, Feb. 2006, pp. 221–230.

[17] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo, "Structured implementation of component based grid programming environments," in *Future Generation Grids*, ser. CoreGRID series. Springer Verlag, Nov. 2005.

[18] C. Zoccolo, "High-performance component-based programming for heterogeneous computing," Ph.D. dissertation, Dept. Computer Science, Univ. of Pisa, 2005.
[19] M. Aldinucci and A. Benoit, "Automatic mapping of ASSIST applications using process algebra," in *Proc. of HLPP2005: Intl. Workshop on High-Level Parallel Programming*, Warwick University, Coventry, UK, July 2005.
[20] M. Aldinucci, A. Petrocelli, A. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo, "Dynamic reconfiguration of Grid-aware applications in ASSIST," Computer Science Department, University of Pisa, Italy, Tech. Rep. TR-05-05, Feb. 2005, submitted to Euro-Par 2005.
[21] E. R. Zayas, "Attacking the process migration bottleneck," in *Proceedings of the 11th ACM Symposium on Operating System Principles*, 1987.
[22] M. Litzkow, "Supporting checkpointing and process migration outside the unix kernel," in *Usenix Winter Conference*, 1992.
[23] E. Godard, S. Setia, and E. White, "Dyrect: Software support for adaptive parallelism on nows," in *Proc. of IPDPS Workshop on Runtime Systems for Parallel Programming*, 2000.
[24] F. Baude, D. Caromel, and M. Morel, "On hierarchical, parallel and distributed components for grid programming," in *orkshop on component Models and Systems for Grid Applications*, 2005.
[25] S. Vadhiyar and J. Dongarra, "Self adaptability in grid computing," *Concurrency and Computation: Practice and Experience*, 2005.
[26] *The Fractal Component Model, Technical Specification*, ObjectWeb Consortium, 2003.