Execution Support of High Performance Heterogeneous Component-Based Applications on the Grid*

Massimo Coppola[1,2], Marco Danelutto[2], Nicola Tonellotto[1,3], Marco Vanneschi[2], and Corrado Zoccolo[4]

[1] Information Science and Technologies Institute, National Research Council
Via G. Moruzzi 1, 56124 Pisa, Italy
[2] Computer Science Department, University of Pisa
Largo B. Pontecorvo 3, 56127 Pisa, Italy
[3] Information Engineering Department, University of Pisa
Via G. Caruso 16, 56122 Pisa, Italy
[4] IAC Search & Media Italia S.r.l.
Corso Italia 58, 56100 Pisa, Italy

Abstract. Application deployment is becoming an increasingly hard task, as complex, component-based Grid applications have to be deployed on heterogeneous and dynamic Grids, interfacing to several different component frameworks and Grid middlewares. We describe the architecture of the Grid Execution Agent (GEA), the deployment and resource brokering tool of the *Grid.it* project. GEA has been designed to ease the deployment of complex Grid applications written in a high-level, structured way. To easily handle different component models over heterogeneous Grid resources, the GEA design exploits multiple levels of abstraction. Our approach allows consistent translation of the high-level requirements from heterogeneous, multi-component applications, to low-level operations over different middlewares. GEA architecture provides a unified interface with services to locate resources, devise initial mapping, and instantiate applications, and it is extensible to new component models. It supports dynamically reconfiguring, self-adapting applications by allowing execution-time resource allocation changes.

1 Introduction

The vision of Computational Grids set forth at the end of last century is becoming reality, at least from the point of view of the raw capability of coordinating Grid resources into executing applications. However, standardization of middleware and practical and efficient programming models for the Grid are still to be

* This work has been supported by: the Italian MIUR FIRB Grid.it project, No. RBNE01KNFP, on High-performance Grid platforms and tools, and the European CoreGRID NoE (European Research Network on Foundations, Software Infrastructures and Applications for Large Scale, Distributed, GRID and Peer-to-Peer Technologies, contract no. IST-2002-004265).

W. Lehner et al. (Eds.): Euro-Par 2006 Workshops, LNCS 4375, pp. 171–185, 2007.

achieved. Thus, the advantages of large Grid computing platforms for several tasks, including collaborative engineering, data exploration, high-throughput computing, and of course distributed super-computing, are still hindered by the difficulty in writing truly portable applications able to exploit dynamic, heterogeneous platforms, as well as to integrate legacy code.

While portals and graphical interfaces allow to manage simple applications and to expose legacy ones as publicly available services, more complex applications designed to benefit from the nature of the Grid platforms still have to be developed exploiting direct interaction with Grid middleware.

Beside the efforts spent in developing middleware systems, the tools provided to deploy and manage the elements of the application do not offer yet a high level of abstraction. Nowadays, the vast majority of applications exploiting Grids are structured as bags of independent jobs, or workflows with simple, file-transfer based interactions.

In the future, complex, multi-disciplinary applications will have to provide an agreed QoS with respect to their fundamental characteristics, e.g. performance, fault tolerance, security. In order to support these requirements more complex and flexible programming models are needed, and applications will have to be able to dynamically alter the set of resources allocated during the execution, and to support multiple interaction protocols with resource management middlewares.

There is general consensus on the adoption of the software component abstraction to simplify the task of programming high performance and distributed applications, especially on Grids. Early examples of this trend are the CCA [1] and GridCCM [2] approaches. Large, international research projects on Grid-aware component models, like CoreGRID and GridCOMP, are a more recent outcome of this trend.

Within the *Grid.it* project we developed the ASSIST [3] structured parallel programming environment to produce software components, and we addressed in the runtime tools the problem of composite application deployment on heterogeneous clusters and Grid resources. The ASSIST environment also supports mixing different kinds of components in the same application (*Grid.it* components, Web services, CCM). These abilities generate the need to integrate different protocols in the run-time for communications, resource query and deployment activities.

In this work we describe the architecture of the *Grid Execution Agent* (GEA), which provides resource brokering and management functionalities for the ASSIST environment. GEA insulates the run-time support of components from the actual Grid middleware. Preliminary versions of GEA have already been introduced in previous works [4,5].

Our main contribution is the design of a Grid application execution framework where a very high-level, abstract description of applications, which is based on software components, is translated into deploy actions using multiple levels of interpretation. The proposed solution exploits plug-in classes to encode the peculiarities, at the different levels of interpretation, of deployment protocols w.r.t. component frameworks, computing processes and supporting middlewares.

The multi-level design of GEA allows easy and seamless configuration of resources and component infrastructures, generally done at run-time, in order (1) to host components from different frameworks, (2) to host components interacting by means of different middlewares, supporting multi-framework integration, (3) to add support for user-transparent deployment of applications on different Grid-middlewares. GEA is thus customizable to support different high-level abstractions interacting with different existing middlewares, supporting HPC Grid applications in large-scale Virtual Organizations, and providing the functionalities of an *Invisible Grid* [3].

The rest of this paper is structured as follows. In Sect. 2 we give a general definition of the Deployment process, with special regard to hierarchical and component-based applications. In Sect. 3 we discuss related work w.r.t. Grid deployment. In Sect. 4 we describe the ASSIST programming environment and the *Grid.it* component model. In Sect. 5 we discuss the approach to application and requirement analysis and translation adopted by GEA, and the overall architecture of the deployment system that results. In Sect. 6 we sum up our contributions and illustrate future work directions.

2 Component Deployment in a Multi-middleware Heterogenous Environment

Our aim is quite general: we want to be able to deploy applications made up of *distributed and parallel components*, which can possibly belong to *different component frameworks*, over *a set of Grid resources* that span a Virtual Organization, possibly encompassing resources managed by *different middleware systems*.

According to our approach, the input of the deployment process includes the application structure, a set of *resource requirements* (fixed constraints on the execution of a single process or components), a set of *QoS models* (analytical expressions of Quality of Service, relating it to the execution parameters of a component) and a set of *contracts* (constraints that the free variables in the application or component model shall satisfy). The initial application deployment will usually involve assembling an overall application QoS model, to balance resource allocation, and decomposing global application contracts into contracts suitable for the single components and modules. Merging contracts and requirements for each component with static knowledge about its implementation, we obtain the information for its initial deployment.

We have to map a non trivial amount of high level information, concerning application structure, deployment requirements and user-expected QoS into a large amount of low-level actions about resource reservation and configuration, process/job mapping and scheduling. Moreover, at run-time more sophisticate models can be used, leading to dynamic changes to the initial deployment choices.

The general problem almost naturally breaks down into levels corresponding to levels of abstraction in the application structure (see Fig. 1).

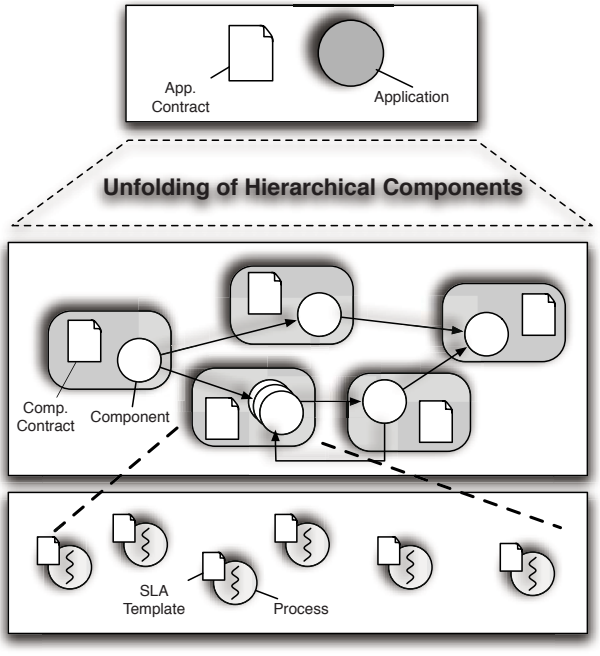


Fig. 1. Abstraction layers for component-based applications execution

Application. An application is a hierarchical composition of components interacting through communication patterns. We need to run it on a Grid while enforcing a user-agreed performance contract, that is a user-dependent specification of the expected behavior of the application at runtime.

Component. The hierarchical composition of components can be unfolded until the whole application is "exploded" to a complex graph of atomic components. We need a standardized way to convey information about the structure and characteristics of every component, as well as models of the runtime behavior of the components and their interactions. Such information is exploited to characterize every component with its own contract, in order to select Grid resources that will host the component, deploy it and control its behavior at runtime.

Process. Each component is made up of several processes (including functional processes and support services). Every process will need its own mapping and scheduling over concrete resources. A Service Level Agreement (SLA) template has to be derived from the component contract for each implementation process, in order to negotiate an agreement with the Grid resources and to globally enforce the QoS required by the user. Moreover, to deploy a component, its processes need to be properly configured to interact.

Middleware. For each single process or job, the Grid middleware used to access the selected resource (e.g. Globus) will generally need a specific set of actions in order to successfully deploy the process.

The deployment activities at each abstraction layer below the first one (component, process, middleware) can be arranged in a workflow, that encodes their dependencies (a partial order over activities), the parameters and the configuration information that each activity needs to transmit to the depending ones.

If we consider each deployment level as the satisfaction of a dependency graph of actions, the overall application deployment is actually the product of the three dependency graphs over components, over processes within them, and over Grid middlewares exploited to access the resources.

To avoid generating large optimization problems, whose solutions would be anyway approximate, we chose not to unroll and flatten the whole application deployment to a single dependency graph of elementary activities.

Our approach instead exploits the hierarchical structure of the application to split the deployment problem into smaller and smaller subproblems. This way we can more easily devise deployment heuristics to reach good initial resource mappings, and it is possible to reuse the same deployment system to perform application adaptation, by deploying locally optimized additional entities (components, processes).

3 Related Work

First-generation deployment mechanisms based on Globus [6] can deploy only sequential jobs and "bag of tasks", that is parallel "uniform" (SPMD) jobs to be executed by homogeneous clusters. Condor-G [7] is a typical example of this kind of approach, as deployment requirements are specified in detail at a very low level of abstraction. Deployment is defined by elementary actions which depend both on the application process structure, and on the middleware. Another obstacle to the aggregation of heterogeneous Grid resources is that Grid middlewares provide in general different APIs, functionalities and servers for resource location, access and management.

These are clearly fundamental issues, which we must solve in order to enhance support for component-based applications, applications with non-trivial structure, e.g. exploiting different frameworks and middlewares, and dynamically adaptive applications, which need execution-time reconfiguration and (re)deployment. A number of systems are currently being developed for the Grid, which aim at solving the mentioned issues and supporting high-level programming languages and environments.

Adage [8] is a tool for Grid deployment whose approach is based on the translation of different kinds of application descriptions, both flat message passing and component-oriented ones, into a common XML format called GADe (Generic Application Description). GADe represents an application as a graph of computing entities, each one made up of processes, and each process containing a set of code entities (components and DLLs). Currently CCM, and MPI translators have been developed which feed with GADe description the deployment planning and execution modules of Adage. GridCCM and CCA translators are in development.

Grid middleware interface in Adage is based on the GAT toolkit [9]. Adage and GEA share a common approach [10] in adopting a core deployment engine independent of application and middleware details, and exploiting a high-level application description language. We differentiate from Adage as our description language (ALDL) allows applications to mix processes and components from different frameworks, and can express dynamic adaptive process networks. While support for automatic translation of descriptions is less developed, GEA can exploit ALDL to manage coallocation of resources over multiple middlewares and frameworks.

The Proactive library [11] is a Java-based solution for parallel and distributed programming. This library provides a programming model and a set of API to develop complex Grid applications. Parallelism in Proactive applications is defined by *Active Objects*, which host application control threads.

To obtain seamless deployment on different runtime environments, Proactive exploits a descriptor-based approach. The *Descriptor Deployment Model* [12] of Proactive is based on three levels of abstraction, (1) Virtual Nodes (VNs) hosting the application specific Active Objects, (2) Java Virtual Machines (JVMs), hosting the application runtime environment, (3) processes, to create and/or acquire JVMs. Virtual nodes are defined in the application code. They are possibly replicated, and instantiated (as Nodes) to run on actual JVMs. JVMs are recruited exploiting information provided by processes.

These mappings are coded in XML Deployment Descriptors, with the target of completely abstracting away from each other the hardware and software runtime configuration, and decoupling application logic from deployment logic.

The approach results in a highly configurable deployment mechanism, which can start new JVMs as needed on local and remote resources. Configuration is left to the Proactive runtime, avoiding any reference to concrete resources in the application code. The drawbacks, and main differences with respect to GEA approach, are that it is difficult to extend the approach to non-Java components, that mapping of application objects to resources is not automatic, and that a first mapping step has to be performed by the user in designing the application, by specifying the mapping of Active Objects to Virtual Nodes.

On the contrary, GEA's Virtual Nodes represent compiler-generated sets of processes, which are not related to the programmer's view of the application, but whose existence is suggested by the implementation of the run-time support.

To the best of our knowledge, KOALA [13] is the only other high-level tool to provide extensive support for coallocation over Grids. KOALA manages reservation and deployment of multi-job applications over the Grid. The job model in KOALA is an executable to be run over a specified number of nodes, taking as input a single data file. The Grid model consists of a set of clusters with homogeneous nodes, interconnected by a known network and each one managed by a compatible local resource manager. These assumptions allow to develop algorithms for multi-site job scheduling taking into account problem parameters like job sizes, input data sizes, available resource loads, network transfer bandwidths and job priority. With respect to KOALA, GEA is less integrated with

resource reservation mechanisms, and it would need more complex job scheduling algorithms to find optimal allocation, due to the Grid model adopted. Nevertheless, GEA can deal with much more detailed resource constraints, allowing to exploit a much more heterogeneous Grid and to satisfy compiler-generated resource requirements in a dynamically evolving environment.

4 The ASSIST Environment Architecture

ASSIST is a high-level parallel programming environment: it provides a structured parallel programming language and a compiler to develop QoS enabled parallel components [3,14]. Basically, applications are described by means of a coordination language, which can express arbitrary graphs of (possibly) parallel modules, which are the basic structural units of applications. ASSIST modules are interconnected by typed streams of data, host portions of sequential code (C, C++, Fortran) and can explicitly define even complex parallel semantics at the module level.

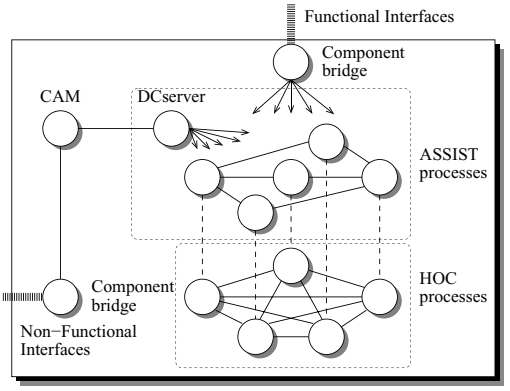
A parallel module (*parmod*) coordinates a set of concurrent activities which are performed by *Virtual Processes* (VPs). We do not fully describe here the ASSIST parallel coordination language [15] or its implementation, we only underline that parmods allow to express parallel computations which are reconfigurable during execution. User-defined code sections within the VPs are seen as the set of atomic computations in the application by the reconfiguration run-time support.

The environment is designed to allow execution of parallel programs over resources that are heterogeneous w.r.t. many characteristics, including CPU architecture, operating system and middleware interface. The compiling tools can also generate *Grid.it* components containing ASSIST code as well as *alien* software resources (e.g. software components from other frameworks).

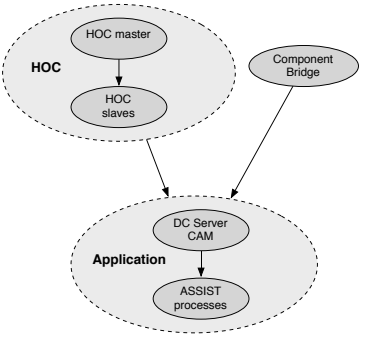
ASSIST/*Grid.it* components [3] are graphs of modules that are explicitly declared as deployment units, and export information and control ports to allow coordinated execution of several of them. *Grid.it* Components expose, among others, *Non-Functional* ports related to QoS control. As shown in Fig. 2a, *Grid.it* native components have a sophisticated internal structure, including different classes of processes devoted to application execution and run-time support,

(1) computational processes (ASSIST processes in the figure),
(2) processes supporting the shared memory abstraction (HOC processes),
(3) manager processes implementing autonomic component behavior (CAM and DCServer), i.e. steering and managing adaptation at the component level,
(4) proxy processes allowing inter-component communication (component bridges).

To deploy even a single component, a workflow of deployment actions is needed, of which we show an example in Fig. 2b. Moreover, *Grid.it* native component can interoperate with Corba/CCM ones (via IIOP-based RPC) [16] and with Web Services (via HTTP/SOAP) [17], forming composite, multi-framework applications. Whether a wrapping approach is adopted, or component bridges



(a) The *Grid.it* distributed component model implementation as a set of processes.



(b) Dependencies at the component level among sets of processes.

Fig. 2. Process structure of a typical Grid.it component, and example of the deployment dependencies among it composing processes

are created, the overall deployment gets more complex. Running such a multi-framework application requires the ability to devise and set up proper support processes for any combination of resource, supporting middleware and supported component framework.

Super-component have been introduced in [18] to describe higher-order components, which can manage parametric graphs of arbitrary components according to a parallel skeleton (i.e. well-known, parametric pattern of parallelism). They provide a fully compositional structure for self-managing Grid applications.

Super-components interact with the resource management and deployment system to manage the life-cycle of their controlled components, and coordinate their overall dynamic reconfiguration. To accomplish this task, they leverage on a compositional self-adapting infrastructure, and on suitable behavioral models corresponding to the parallel skeletons the specific super-component implements.

At any moment during an ASSIST application run, modules and components can be assigned a new QoS contract, e.g. specifying a performance, security or fault tolerance requirement. In order to fulfill the contracts, the component framework continuously adapts component configurations, in terms of parallelism degree, and process mapping [19]. This means having a **progressive, dynamic deployment process** where portions of the application are re-deployed in order to meet a specific QoS target.

The adaptation mechanism relies on automatic user code instrumentation, and on a **hierarchy of Application Managers** [3] exploiting knowledge about the application structure and the run-time implementation. The hierarchy of managers operating at different levels in the application structure is reflected in the connections among the Non-functional ports of modules, components, and super-components. Eventually, the whole execution is steered by the top-level Application Manager (AM) component. Semantics and protocols of these

interactions are out of the scope of the paper, but we point out that some dynamic effects of resource management have non-local impact which need proper handling in the management hierarchy (e.g. load balancing may need computing resources in excess to be re-allocated to some seemingly unrelated part of the application).

From the ASSIST point of view, the GEA is a component of the environment run-time support, the Grid Abstract Machine, as it manages the resource allocation at all levels. ASSIST super-component managers leverage the GEA when deploying new component(s), and the component and module run-time support for reconfiguration (CAM and MAM entities) can contact GEA whenever resources are needed to spawn new processes in order to satisfy performance contracts at the module level.

As a final remark, compositional, hierarchical component models (e.g. an implementation of Fractal in Java or C++) also need an algorithmic way to break down the overall application description into the set of descriptions of its components, and in particular to **project the application QoS specification over that of components**, in order to find appropriate resources to deploy each component. This phase of "requirement unfolding" can happen outside of, or as part of the deployment workflow. Current approach in *Grid.it* exploits the model embedded in super-components. Transformation from application to component-level requirements is recursively performed by the Application Manager, which directly controls the unfolding step of deployment. As a different approach, a compositional performance model for launch-time mapping has also been developed [20], which is suitable to devise a good initial resource allocation and speed up the deployment of the whole application. Such a model can be produced and evaluated during deployment, for instance implementing it within a particular Component Translation Engine Plug-in (see Sect. 5.2).

5 Grid Execution Agent Design

The Grid Execution Agent (GEA) is the automatic tool developed within the *Grid.it* project to seamlessly search resources for, deploy, and run complex component-based Grid applications. The ASSIST/*Grid.it* environment targets high-performance, data/computation intensive, and distributed applications. GEA is designed to be a high-level resource management system, handling all the low-level interaction with multiple Grid middlewares and with the code providing dynamic adaptation. The Core of the Deployment cycle, as shown in Fig. 3 in the inner box, follows the outline presented in [4]. We recall it in short in the next section, before discussing its extension to a dynamic and multi-middleware scenario.

5.1 Core Deployment Cycle

The input of the cycle is a description of the entities to deploy in a general format, the Application Level Description Language (ALDL). This XML dialect can encode the requirements for all the deployment entities, ranging from high

level performance specifications for components to concrete constraints on target architectures for processes.

ALDL descriptions contain different types of information:

- static resource constraints and dynamic constraints – e.g. constraints related to quantities that do not vary over time, like peak performance of a resource, as opposed to those related to varying features, like available computation bandwidth, which depends on resource load.
- hardware and software constraints – expressing the specific need of architectures, operating systems, support or application libraries to be available to an entity at its execution site.
- aggregate constraints – specifying constraints on groups of entities. Most notably, constraints over communication networks (security, reachability) and over sets of processes which employ specific common resources or launch protocols (e.g. name services or fault-tolerant communication schemes).

GEA, starting from the ALDL description, automatically performs resource discovery and selection, handles data and executable file transfers. Different GEA modules perform successive steps of translation of high-level specifications into deploy actions.

1. The ALDL description is parsed and an internal representation of the graph of tasks is generated, annotated with specific requirements and constraints.
2. From the internal representation, resource queries are computed, which aim at locating a set of resources satisfying all the static constraints.
3. Resource queries are executed exploiting the middleware.
4. A subset of the resources is selected, also exploiting information related to dynamic constraints.
5. The graph of processes is mapped over the resources. This can result in mapping cooperating entities over independently managed resources, thus triggering coallocation in the following phase.
6. Finally, each entity is executed on the corresponding resource through the middleware. This means e.g. staging and executing process code, configuring its execution environment, or deploying a specific network configuration to ensure a stated goal of communication QoS.

Actually, the discovery and mapping phases can loop until a set of resources is found that allow to map the whole entity according to its execution constraints. As a result of the deployment process, different parts of the ALDL description are filtered, instantiated and translated into the appropriate forms to be enacted on the middleware used by each part of the Grid computing platform.

5.2 A Modular Multi-middleware Architecture

The core cycle described in the previous section deploys applications over a heterogeneous Grid, can exploit different middleware to access resources, but it coordinates them in the execution of a single program.

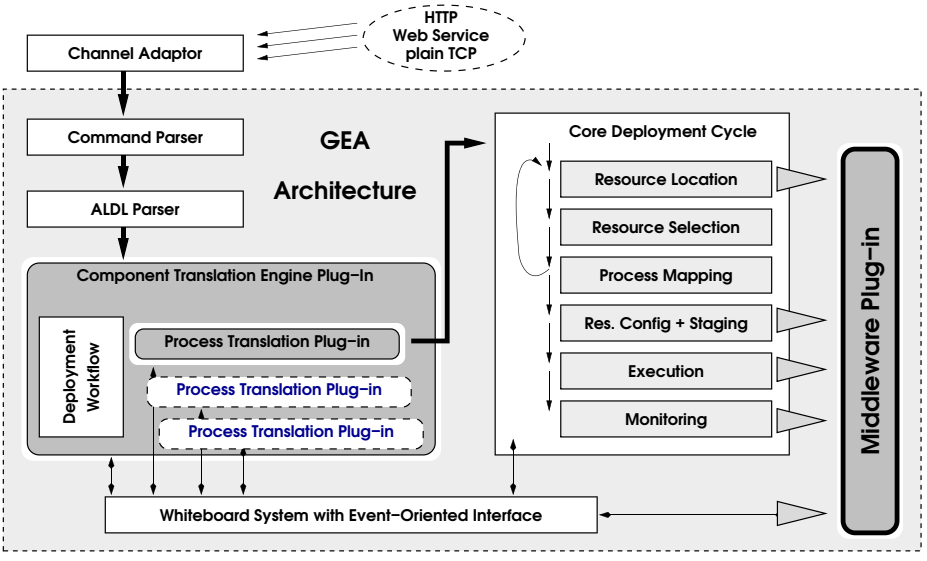


Fig. 3. GEA high-level architecture

As ASSIST was being developed into a component model, additional requirements were put onto GEA.

Separate/Dynamic Deployment. Applications are made up of multiple components, and components are separate units of deployment. GEA has to fully behave like a server, allowing to launch multiple components (eventually from different applications) and to manage each one separately over possibly overlapping portions of the Grid.

Dynamic adaptation. *Grid.it* components can ask for and free resources (e.g. processing nodes) at run-time, so they need access to deployment functions.

Access Protocols. GEA functionalities have to be accessible through different protocols (plain TCP sockets, HTTP, Web Services), in order to exploit them easily across the Grid.

Flexibility w.r.t. Component Models. *Grid.it* applications can exploit components and services from other frameworks (CCM, Web Services), which need to be deployed and accessed with their own protocols. Moreover, it is a key feature to ease experimentation with various implementations of a component model.

Higher Scalability. Provision to the user of a single point of access to the whole Grid must not bring unneeded centralization and impair deployment scalability.

Crossing of Domain Boundaries. GEA should be able to operate at the boundary of a network and provide to the outside a unified access abstraction, independently of any Grid middleware present within the network.

The resulting extended architecture in Fig. 3 takes into account all these issues and builds upon the core cycle used in the first design of GEA. It plans and enacts the deployment workflow of *Grid.it* components, starting in the proper order the server processes and service daemons needed by any component, as well as the processes actually performing the computation.

A key point is that, to provide flexibility in experimenting with component models, types of processes and diverse middlewares, GEA has been extended via plug-in classes that implement different component setup workflows, process launch protocols and interfaces to middleware primitives.

Channel Adaptor. A channel adaptor module is used in GEA to support multiple input protocols and control interfaces. For instance, the GEA server interacts with the CAM component manager via TCP, with the user through a set of command-line tools, provides Web Services / HTTP as a standard interface. Authorization mechanisms (e.g. local or GSI authentication) can be used to restrain the access to some of the adaptors, and, as different interfaces can expose only part of the full set of GEA commands, to provide different authorization levels.

Command Parser. The GEA command parser supports commands to manage the life-cycle of deployed entities (from providing them in archival form, to monitoring their termination) and to control the configuration of the GEA server. provide components in form of archives, to deploy component instances, to dynamically add new resources, to monitor component termination, to quit the GEA server or to reload its static configuration information (e.g. addresses of dynamic Grid information services). The command parser is also in charge of managing multiple sessions: each component's ALDL is kept linked to a session identifier, which is a handle to monitor and steer the component deployment and the set of allocated resources. The ALDL representation of each component must be kept (1) to allow caching of component archives and easily instantiate multiple copies of a component, (2) to simplify handling of dynamic adaptation, as we can deploy additional processes within a component by referring to their identifiers in the component description. The parser manages multiple command session and internally caches ALDL representation of components, in order to ease creating multiple instances of a component, and to allow partial redeployment of a running component.

ALDL Parser. The ALDL parser has been extended (w.r.t. [4]) to allow expressing explicit co-allocation and superposition of processes. A concept of *virtual node* is used (close to that of Proactive) which is the abstraction of a physical resource. Processes of different types are mapped to virtual nodes, which are the units of low-level resource mapping. In order to subsequently map virtual nodes on resources, process constraints are gathered and summed up with the proper aggregation function (e.g. sum, for memory requirements, union, for requirements over available libraries, and so on).

Component Translation Engine. The Component Translation Engine is actually the highest level plug-in, transforming a component ALDL

specification into a network of process dependencies. The deployment workflow, hard-coded as a Java class in the plug-in, fulfills the dependency graph among different types of processes in a given component model,

Process Translation Plug-in. Process Translation Plug-ins are a set of mid-level translator classes, associated with the types of processes we need to start. Each of these plug-ins can add special constraints over the resources to select (e.g. having a public IP address), it can exploit information from previously configured/deployed processes and service daemons, and it knows the protocols to configure and deploy one type of processes (e.g. ASSIST DCserver and ASSIST application processes). Translated requirements of all processes within a component are produced by the appropriate plug-ins, before starting the actual deployment process.

Core Deployment Cycle. As reported, the core deployment cycle has been adapted from the previous GEA architecture, with all requirements gathered from all virtual nodes first, and then satisfied all at the same time as described in Sect. 5.1. The actual deploy order of processes (configuration, staging and execution phases) in governed by the dependencies explicitly introduced by the Component Engine and Process Translation plug-ins.

Middleware Plug-in. This is a low-level set of classes, one for each middleware supported, exposing a set of primitives for the basic operations of all the steps defined in the Core Deployment Cycle (resource location, selection, mapping, staging and so on). Existing plug-ins support Globus managed resources (using MDS as information repository), as well as SSL-based access to clusters and local networks (XML static configuration files are used in place of MDS). In some cases GEA extends the functionalities of the middleware, e.g. to provide status monitoring for resources accessed via simple SSL.

Event System and GEA Whiteboard. A communication module is used as a scratch-pad interface to allow uniform parameter management to plug-ins of all levels. Different process types and different instances within a component in general need to exchange and propagate synchronization information, service addresses and other execution parameters (e.g. CCM processes are run after their naming service is known and it is up). The whiteboard implementation manages several kinds of events (TCP and HTTP communications, process termination, monitoring information) and uses them to trigger deployment dependencies by means of callback functions that notify registered plug-ins.

6 Conclusion and Future Work

We have presented the architecture of the GEA deployment tool developed within the *Grid.it* project. The presented extension to the deployment cycle developed in the previous implementation [4] allows greater flexibility and easier customization of the application model, with respect to the types of component we can actually deploy. The ASSIST application model, the *Grid.it* component

model and the launch and configuration procedures for component support elements are all boxed into separate plug-ins.

We are currently working on the integration of different component and application models within GEA, including CCM components and Web Services (within the *Grid.it* project) and POP C++ applications (within the CoreGRID NoE). The new implementation matches with the extensions of the ALDL language, which we just mentioned in this work, to cope with new mapping constraints and with contract specifications.

Currently, we are still working on the ALDL language to improve expressiveness w.r.t. QoS contract specification for applications and components. A technique that allows to derive component constraints from application ones has already been devised [20], which can be used to develop a Component Translation Plug-in to deal with a fully hierarchical component model (e.g. Fractal) to devise an optimal initial application mapping starting from the ALDL specification and the description of the available resources.

Both the implementation of a hierarchy of GEA servers for distributed deployment, and of more scalable deploy protocols for very large Grids, are in our future plans. We have made preliminary experiments in this direction, exploiting hierarchical and distributed communication schemes in overlay networks of servers. We are going to exploit the flexibility of the GEA architecture to integrate these prototypes as a new channel adaptor and dedicated low-level plug-ins.

References

1. Armstrong, R., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B.: Toward a common component architecture for high-performance scientific computing. In: 8th IEEE International Symposium on High-Performance Distributed Computing. (1999)
2. Pérez, C., Priol, T., Ribes, A.: A parallel CORBA component model for numerical code coupling. In: GRID 2002 : Third International Workshop, Springer (2002)
3. Aldinucci, M., Campa, S., Coppola, M., Danelutto, M., Laforenza, D., Puppin, D., Scarponi, L., Vanneschi, M., Zoccolo, C.: Components for high performance Grid programming in Grid.it. In: Proc. of the Workshop on Component Models and Systems for Grid Applications. CoreGRID series. Springer (2005)
4. Danelutto, M., Vanneschi, M., Zoccolo, C., Tonellotto, N., Baraglia, R., Fagni, T., Laforenza, D., Paccosi, A.: HPC Application Execution on Grids. In Getov, V., Laforenza, D., Reinefeld, A., eds.: Future Generation Grids. CoreGRID. Springer (2006) Dagstuhl Seminar 04451 – Nov. 2004.
5. Adami, D., Giordano, S., Repeti, M., Coppola, M., aforenza, D.L., Tonellotto, N.: Design and Implementation of a Grid Network-Aware Resource Br oker. In Fahringer, T., ed.: Parallel and Distributed Computing and Networking 2006. ACTA Press (2006)
6. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Int. J. of Supercomputer Applications and High Performance Computing **11** (1997) 115–128
7. Frey, J., Tannenbaum, T., Foster, I., Livny, M., Tuecke, S.: Condor-G: A Computation Management Agent for Multi-Institutional Grids. In: Proceedings of the 10th IEEE Symp. on High Performance Distributed Computing (HPDC10), San Francisco, California, IEEE (2001)

8. Lacour, S., Pérez, C., Priol, T.: Generic application description model: Toward automatic deployment of applications on computational grids. In: 6th IEEE/ACM International Workshop on Grid Computing (Grid2005), Seattle, WA, USA, Springer-Verlag (2005)
9. Allen, G., et al.: Enabling Applications on the Grid – A GridLab Overview. International Journal of High Performance Computing Applications **17** (2003) 449 – 466 Special issue on Grid Computing: Infrastructure and Applications.
10. Coppola, M., Danelutto, M., Lacour, S., Pérez, C., Priol, T., Tonellotto, N., Zoccolo, C.: Towards a Common Deployment Model for Grid systems. To appear in CoreGRID series. (2005)
11. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Deploying, Composing, for the Grid. In: Grid Computing: Software Environments and Tools. Springer-Verlag (2006)
12. Baude, F., Caromel, D., Mestre, L., Huet, F., Vayssière, J.: Interactive and descriptor-based deployment of object-oriented grid applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, IEEE Computer Society (2002) 93–102
13. Mohamed, H., Epema, D.: The Design and Implementation of the KOALA Co-allocating Grid Scheduler. In Sloot, P.M.A., Hoekstra, A.G., Priol, T., Reinefeld, A., Bubak, M., eds.: Advances in Grid Computing - EGC 2005: European Grid Conference. Volume 3470 of LNCS. (2005) 640–650
14. Aldinucci, M., Coppola, M., Danelutto, M., Vanneschi, M.V., Zoccolo, C.: ASSIST as a research framework for high-performance Grid programming environments. In Cunha, J.C., Rana, O.F., eds.: Grid Computing: Software environments and Tools. Springer (2005) 1–32 (To appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, 2004).
15. Vanneschi, M.: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing **28** (2002) 1709–1732
16. Magini, S., Pesciullesi, P., Zoccolo, C.: Parallel software interoperability by means of CORBA in the ASSIST programming environment. In: Kosch, H., Böszörmeny, L., Hellwagner, H.(eds.) Euro-Par 2003. LNCS, vol. 3648, pp. 679–688. Springer, Heidelberg (2004)
17. Aldinucci, M., Danelutto, M., Paternesi, A., Ravazzolo, R., Vanneschi, M.: Building interoperable grid-aware ASSIST applications via WebServi ces. In: PARCO 2005: Parallel Computing, Malaga, Spain (2005)
18. Aldinucci, M., Bertolli, C., Campa, S., Coppola, M., Vanneschi, M., Veraldi, L., Zoccolo, C.: Self-Configuring and Self-Optimising Grid Components in the GCM model and their ASSIST Implementation. In: Joint Workshop on HPC Grid Programming Environments and Components (HPC-GECO/CompFrame). (2006)
19. Aldinucci, M., Petrocelli, A., Pistoletti, E., Torquati, M., Vanneschi, M., Veraldi, L., Zoccolo, C.: Dynamic reconfiguration of grid-aware applications in ASSIST. In Cunha, J.C., Medeiros, P.D., eds.: 11th Intl Euro-Par: Parallel and Distributed Computing. Volume 3648 of LNCS., Lisboa, Portugal, Springer (2005) 771–781
20. Tonellotto, N., Zoccolo, C.: Characterization of the performance of ASSIST programs. Technical Report TR-0007, CoreGRID - Network of Excellence (2005)