Lightweight Grid Platform: Design Methodology.

Rosa M. Badia[5], Olav Beckmann[2], Marian Bubak[6], Denis Caromel[3], Vladimir Getov[4], Ludovic Henrio[3], Stavros Isaiadis[4], Vladimir Lazarov[1], Maciek Malawski[6], Sofia Panagiotidi[2], Nikos Parlavantzas[3], Jeyarajan Thiyagalingam[4]

[1] Institute for Parallel Processing, Acad. G. Bonchev
[2] Department of Computing, Imperial College London
[3] CNRS URA 1376, INRIA, 2004 Rt. des Lucioles, BP 93,F-06902 Sophia Antipolis, Cedex, France.
[4] Harrow School of Computer Sciences, University of Westminster
[5] Departament d'Arquitectura de Computadors (DAC), Universitat Politecnica de Catalunya
[6] Academic Computer Centre – CYFRONET, Krakow, Poland.

Lightweight Grid Platform: Design Methodology

Rosa M. Badia[5], Olav Beckmann[2], Marian Bubak[6],
Denis Caromel[3], Vladimir Getov[4], Ludovic Henrio[3],
Stavros Isaiadis[4], Vladimir Lazarov[1], Maciek Malawski[6],
Sofia Panagiotidi[2], Nikos Parlavantzas[3], Jeyarajan Thiyagalingam[4]

[1]*Institute for Parallel Processing,*
Acad. G. Bonchev Str., Bl. 25-A, Sofia, Zip 1113, Bulgaria.
[2]*Department of Computing, Imperial College London, London SW7 2AZ, U.K.*
[3]*CNRS URA 1376, INRIA, 2004 Rt. des Lucioles, BP 93,F-06902 Sophia Antipolis, Cedex, France.*
[4]*Harrow School of Computer Sciences, University of Westminster,*
Watford Road,Northwick Park,Harrow HA1 3TP, United Kingdom
[5]*Departament d'Arquitectura de Computadors (DAC),*
Universitat Politècnica de Catalunya, Campus Nord - Mòdul D6,
C/ Jordi Girona, 1-3, E-08034, Barcelona, Spain.
[6]*Academic Computer Centre – CYFRONET, Nawojki 11,30-950 Kraków, Poland.*

CoreGRID Technical Report
Number TR-0020
January 25, 2006

Institute on Grid Systems, Tools and Environments

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Lightweight Grid Platform: Design Methodology

Rosa M. Badia[5], Olav Beckmann[2], Marian Bubak[6],
Denis Caromel[3], Vladimir Getov[4], Ludovic Henrio[3],
Stavros Isaiadis[4], Vladimir Lazarov[1], Maciek Malawski[6],
Sofia Panagiotidi[2], Nikos Parlavantzas[3], Jeyarajan Thiyagalingam[4]

[1]Institute for Parallel Processing,
Acad. G. Bonchev Str., Bl. 25-A, Sofia, Zip 1113, Bulgaria.
[2]Department of Computing, Imperial College London, London SW7 2AZ, U.K.
[3]CNRS URA 1376, INRIA, 2004 Rt. des Lucioles, BP 93,F-06902 Sophia Antipolis, Cedex, France.
[4]Harrow School of Computer Sciences, University of Westminster,
Watford Road,Northwick Park,Harrow HA1 3TP, United Kingdom
[5]Departament d'Arquitectura de Computadors (DAC),
Universitat Politècnica de Catalunya, Campus Nord - Mòdul D6,
C/ Jordi Girona, 1-3, E-08034, Barcelona, Spain.
[6]Academic Computer Centre – CYFRONET, Nawojki 11,30-950 Kraków, Poland.

CoreGRID TR-0020

January 25, 2006

Abstract

Abstract

Design aspects of existing and contemporary Grid systems were formulated with the intention of utilising an infrastructure where the resources are plentiful. Lack of support for adaptivity, reconfiguration and re-deployment are some of the shortcomings of existing Grid systems. Absence of capabilities for a generic, light-weight platform with full support for component technology in existing implementations has motivated us to consider a viable design methodology for a light-weight Grid platform. In this paper we outline the findings of our preliminary investigation.

Keywords Generic Grid, Light-Weight platform, components technology

1 Introduction

Grid technology has the potential to enable coordinated sharing and utilisation of resources in large-scale applications. However, the real benefits are greatly influenced and restricted by the underlying infrastructure.

Existing, contemporary Grid platforms are feature-rich, such that the requirements of end users are a subset of the available features. This design philosophy introduces considerable software and administrative overheads, even for relatively simple demands. The absence of a truly generic, lightweight Grid platform with full support for component technology as well as adaptivity, reconfiguration and dynamic deployment in current Grid systems has motivated us to consider a viable design methodology for engineering such a Grid platform.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

In [27], Thiyagalingam *et.al.* set out the design principles for designing a lightweight Grid platform. In this paper, we extend their techniques to design a lightweight, generic Grid platform, by carefully analysing requirements and enabling technologies along with special attention to component technology. The key to our design philosophy is dynamic, on-demand pluggable component technology through which we hope to add and remove features on demand.

The rest of this paper is organised as follows: In Section 2 we review some existing component models. Section 3 justifies the requirements for a generic, lightweight Grid platform while Section 4 evaluates existing and enabling technologies. Section 5 includes some use-case scenarios to illustrate the needs and requirements of the platform and Section 6 concludes the paper with directions for further research.

2 Component Technologies

At least the following three different component models influence our design:

- Common Component Architecture (CCA) [10]
- Fractal Component Model [6]
- Enterprise Grid Alliance Reference Model [11]

In the Common Component Architecture (CCA) [10], components interact using ports, which are interfaces pointing to method invocations. Components in this model define “provider-ports” to provide interfaces and “uses-ports” to make use of non-local interfaces. The enclosing framework provides support services such as connection pooling, reference allocation, etc. Dynamic construction and destruction of component instances is also supported along with local and non-local binding. An interface description language (known as Scientific Interface Description Language, SIDL [26]) may be used to specify the interfaces and associated constraints which are then later compiled using a dedicated compiler (SIDL Compiler) to generate source code in a chosen programming language. These features provide seamless runtime interoperability between components. However, CCA does not strictly specify component composition and control mechanisms.

The Fractal Component Model [6] proposes a typed component model in which a component is formed from a controller and a content, and Fractal components may be nested recursively inside the content part. The control part provides a mechanism to control the behaviour of the content either directly or by intercepting interactions between Fractal components. The recursive nesting, sharing and control features support multiple configurations. Components have well-defined access points known as *interfaces*, which could either be client- or server-interfaces. Components interact through interfaces using *operations*, which are either one-way or two-way operations. The operation type determines the flow of operation requests and the flow of results, i.e. one-way operations correspond to operation requests only, whereas two-way operations correspond to operations with results being returned. As the controller part may be used to manipulate the behaviour of the content part, the various composition operations can be formulated on-demand. This feature, combined with sophisticated binding technology, may be used to re-configure components and compositions dynamically.

The Enterprise Grid Alliance provides a reference model [11] with the intention of adopting Grid computing related technologies within the context of enterprise or business. The model, which is aligned with industry-strength requirements, classifies the components, which may include hardware resources, into layers. Components can be associated with component-specific attributes to specify dependencies, constraints, service-level agreements, service-level objectives and configuration information. One of the key features that the reference model suggests is the life-cycle management of components which could be governed by policies and other management aspects. Unlike other two models, EGA is not a strong component model; yet, we have included it here for its support for component life-cycle management.

3 Requirements for a Generic Lightweight Platform

The complete set of features and requirements for any piece of software evolves over time. However, a reasonable set of requirements and features can be derived by analysing the requirements of end-users and similar frameworks. Further, in realizing the design of the platform, we would like to utilise techniques available in existing work. For this purpose, we have analysed the following relevant frameworks.

- MOCCA/H20 [19]
- ProActive (and other realizations of Fractal) [12]
- CORBA [22]
- ICENI [18]
- Ibis [25]
- GRID superscalar [3]
- Enterprise Grid Alliance Reference Model [11]

Although some of these are not platform-level frameworks, they do support some key technologies which are necessary either as part of a Grid platform or as an enabling technology for designing a Grid platform. For instance, ProActive supports very strong component-oriented application development; Ibis provides an optimisation framework for communication-bound programs; Grid Superscalar facilitates improving the performance of a certain class of applications by identifying specific data-flow patterns; MOCCA, a partially implemented lightweight platform, supports modular development. We discuss these enabling technologies in Section 4.

Following the analysis of these key technologies, we propose the following key requirements for a generic, lightweight platform.

1. **Lightweight and generic**

   Grid computing has the potential to address grand challenges, starting with vehicle design to analysing financial trends. However, currently, its benefits are confined to a computing environment where the resources are plentiful. Traditional design methodologies for Grid systems, where systems are expected to be feature-rich, do not produce generic Grid platforms.

   Primarily, a Generic Grid Platform should be lightweight with minimal but essential features such that it could be scaled by adding new features as required. Such a property would permit us to enable Grid technologies being utilised from consumer devices to enterprise data-centres.

2. **Static and dynamic metadata**

   In complex distributed computing environments, metadata plays an important role. Especially in component-based environments, it is often imperative to be able to extract metadata information from components in order to ensure efficient composition of components, satisfy quality of service requirements and provide the building blocks for the dynamic properties of the platform (reconfigurability, adaptability). For optimal application composition it is necessary to hold information about each available component. Static metadata can provide information pertaining to implementation, version, compatibility issues, performance characteristics, restrictions, accounting details and alike.

   At the other end we have dynamic metadata information: information pertaining to dynamic properties of components and resources. Keeping track of dynamic properties of components and resources is vital for satisfying quality of service and other service-level agreements. Further, dynamic metadata can be used for efficient component optimisation, checkpointing, recovery from failures, logging, accounting and reporting. Dynamic metadata can go beyond isolated components, and cover the application composition as a whole in order to support cross-component optimisation, application steering, run-time dynamic workload balancing etc.

3. **Dynamic deployment of components**

   There are many reasons why the platform should deploy components dynamically, including reaction to changes and demands in the system. This is possible, only if the platform is capable of introducing, replacing and removing components dynamically with minimal disruption.

4. **Reconfiguration and adaptivity**

   The platform should realistically model and synthesise resources in order to install or un-install dynamically additional features and services. Further, appropriate reaction to environmental conditions with the right exploitation of modelling, synthesis and deployment of services is also a necessary feature of the platform to guarantee resilience to failures.

This is essentially a form of reconfiguration or the ability of the platform to self-organise itself to agree with the QoS issues, service-level agreements and service-level objectives. In supporting reconfiguration and adaptivity, the platform may utilise rules, embedded-knowledge and knowledge gathered across runs.

5. **Support for both client/server and P2P resource sharing**

   In the traditional client/server model of resource sharing, a broker module (or a querying module) performs match-making between user requirements and resources. In contrast, in a decentralised system, providers and consumers interact with more freedom without the intervention of brokering modules, i.e. in a peer-to-peer fashion. While a regulated centralised access mechanism guarantees enforcement of fair policy, a peer-to-peer mechanism reduces associated overheads and improves response time and performance. Further, in an ad-hoc and mobile environment, registration activities may introduce unnecessary delays. In contrast, a peer-to-peer scheme does not require any such mechanism at the central level. We hypothesise that the platform should support both of these schemes to enable context-based support for resource sharing.

6. **On-demand, provider-centric service provision**

   The platform should support on-demand creation of services when needed by clients. However, making service provision more provider-centric ensures that the platform is freed from complex resource modelling and binding issues. In a provider-centric model, the provider enables service provision. This essentially frees the platform from performing unnecessary negotiations and coordination tasks.

7. **Minimal but sufficient security model**

   Maximised security, performance and simplicity of the platform are contradictory goals in design. Though the minimal security model may offer acceptable performance and may result in a lightweight platform, it can in practice be challenging to quantify the right level of “minimal security”. Security requirements are often context-based. For example, in a trusted or isolated network, security measures can be bypassed in favour of performance and simplicity. In a collaborative network, such as the Internet, it is inevitable that security measures are tightened with minimal concern over performance issues. We intend to include support for single sign-on and delegation of credentials and mechanisms required in resource sharing environments which may span multiple administrative domains and pluggable support for any additional security features. This approach guarantees that the security model may evolve with context.

8. **Binding and coordination**

   Resources should be configurable by pluggability and reconfigurability of the platform, thus making possible the scenarios available in H2O [16]. The roles of resource provider, component deployer and client should be separated, but they may possibly overlap. The platform should not mandate a specific mechanism for coordination and matching of users and providers. This should be left for pluggable discovery and brokering components. The presence of a centralised coordination point enables effective binding of resources, providers and users. However, such a centralised point can be a bottleneck in a loosely federated environment with no control. This would urge us to consider technologies for binding of resources, providers and users through a decentralised scheme with limited negotiation for provision, utilisation and coordination of entities.

9. **Additional services**

   The platform should be able to incorporate additional services as requested by the environment. For example, a network environment may opt to bill the users during peak time (utility accounting), provide additional smarter discovery protocols at a small charge, automated backup services etc.

10. **Distributed management**

    Distributed but coordinated management functionality is the heart of the platform operation. These functionalities may including life-cycle management of components, workflows, meta-data and utilisation of meta-data.

4 Evaluation of Existing Technologies

- **MOCCA/H2O**: MOCCA [19] is a lightweight distributed component platform, an implementation of the CCA framework built on top of the Java-based H2O resource sharing platform. MOCCA uses H2O [16] as a mechanism for creation of components on remote sites and uses RMIX [17] for communication between components.

MOCCA takes advantage of the separation of the roles of resource provider and service deployer in H2O. Components in MOCCA can be dynamically created on remote machines. H2O kernels, where components are deployed, use the Java security sandbox model, giving a secure environment for running components. The extensible RMIX communication library allows using various protocols for communication, such as JRMP or SOAP, and also pluggable transport layers, including TCP, SSL, and JXTA [15] sockets for P2P environment.

- **ICENI and ICENI II**: ICENI [20, 13] is a Grid middleware infrastructure which includes methods for both resource management and efficiently deploying applications on Grid resources. The design philosophy of ICENI is based on high-level, component-based software construction, combined with declarative metadata that is used by the middleware to achieve effective execution. ICENI II [21] is a natural semantic evolution of ICENI, maintaining the architectural design of the original ICENI, but overcoming weaknesses in the current implementation, such as the implementation of ICENI on top of Web Services, decomposition of ICENI architecture into a number of separated composable toolkits and reduction of the footprint of ICENI on resources within the Grid.

- **ALiCE**: ALiCE [1] is a lightweight Grid middleware which facilitates aggregation and virtualization of resources within an intranet and leveraging sparse resources through the Internet. The modularised, object-oriented nature of its implementation supports possible extensions and varying the levels of QoS, monitoring and security. The ALiCE architecture consists of multiple layers with the lowest, core layer providing resource discovery and system management using Java technologies. The second level layer, relying on the lowest layer, supports application development and deployment. The ALiCE runtime system consists of consumer, resource broker and a producer and task-farm manager which deploys and executes applications.

- **IBIS**: Ibis [25] is a Java-based Grid programming environment, aiming to provide portability, efficiency and flexibility. Ibis offers such programming models as traditional RMI (Remote Method Invocation), GMI for group communication, RepMI for replicated objects and Satin for solving problems using divide-and-conquer method. These components of Ibis are placed on top of the Ibis Portability Layer (IPL), which allows various implementations of underlying modules, such as communication and serialisation, monitoring, topology discovery, resource management, and information services. IPL allows runtime negotiation of optimal protocols, serialisation methods, and underlying grid services, depending on the hardware and software configuration and requirements from higher layers.

  Ibis focuses on various performance optimisations, to overcome the known drawbacks of Java. The optimisations include the serialisation of objects in RMI, avoiding of unnecessary copying of data during communication, and possibility of using native communication libraries e.g. for Myrinet.

- **CORBA**: Common Object Request Broker Architecture (CORBA [22]) is a middleware specification for large-scale distributed applications. An application in the CORBA architecture is composed of objects and the description of operations and functionalities of each and every object is utilised for providing support at the architecture level. Interface descriptions are used for communicating objects, transporting data and marshaling/unmarshaling methods calls. The IDL (Interface Description Language) definitions are language-independent, through mappings from a chosen programming language. The interfaces are compiled and mapped to the underlying programming language with a compiler provided by the ORB (Object Request Broker) system, which is the key to CORBA's interoperability. Method invocations on objects are handled transparently by the ORB, providing maximum abstraction. To capture dynamically and provide information regarding new objects, the ORB model provides a Dynamic Invocation Interface (DII) , which unifies the operations to all instances of an object. With DII, clients can construct the invocations dynamically by retrieving the object IDL interface from the registry.

- **GRID Superscalar**: GRID Superscalar [3] is an Grid-unaware application framework focused on scientific applications. The definition of Grid-unaware applications in the framework of GRID Superscalar are those applications where the Grid (resources, middleware) is transparent at the user level, although the application will be run on a computational Grid. The key for GRID Superscalar applications is the identification of coarse grain functions or subroutines (in terms of CPU consumption) in the application. Once these functions or subroutines (called tasks in the GRID Superscalar framework) are identified, the GRID Superscalar system is able to detect at runtime data dependencies and the inherent concurrency between different instances of the tasks. Therefore, a data-dependence task graph is dynamically built, and tasks are executed on different resources on the Grid. Whenever possible (because data-dependencies and available resources allow) different tasks are executed concurrently, increasing application performance.

The input codes for GRID Superscalar are sequential applications written in an imperative language, where a small number of GRID Superscalar API calls has been added. Another input that the user should provide is the IDL file, where the coarse grain functions/subroutines are identified by the user specifying their interface. A code generation tool uses the IDL file to generate all the remaining files so that the application can be run on a Grid environment. This is combined with the deployment centre, which is graphical interface that enables to check the grid configuration and to automatically deploy the application in the grid. Optionally the user can also specify determined requirements of the tasks (resource, software, hardware) in a constraint specification interface. These requirements are matched at runtime by the GRID Superscalar library and the best resource that meets the requirements is selected to execute each task.

- **ProActive**: ProActive is a 100% Java library for parallel and distributed computing. ProActive is based on a meta-object protocol (MOP): objects and method calls are reified [9]. ProActive allows to build Grid applications by composing them from existing components, for instance by programming (using scripting or compiled languages), and also to build the individual components.

  As ProActive is built on top of the Java standard APIs, mainly Java RMI and the Reflection APIs, it does not require any modification to the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine. Additionally, the Java platform provides dynamic code loading facilities, very useful for tackling complex deployment scenarios.

  A distributed or concurrent application built using ProActive is composed of a number of medium-grained entities called *activities*. Each active object has one distinguished element, the *active object*, which is the only entry point to the activity. Each activity is single threaded and decides in which order to serve the incoming method calls. Method calls sent to active objects are asynchronous with transparent *future objects* and synchronisation is handled by a mechanism known as *wait-by-necessity* [8]. There is a short rendezvous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee, thus ensuring causal ordering of communications. ProActive also features mobility, security [2], and group communication [4].

  ProActive components provide a distributed implementation of Fractal components. More precisely, a ProActive component follows the characteristics below:

  - It provides a set of server ports as defined in Fractal [7] (Java Interfaces)
  - It possibly defines a set of client ports (Java attributes if the component is primitive)
  - It can be of three different types :
    1. primitive : defined with Java code implementing provided server interfaces, and specifying the mechanism of client bindings.
    2. composite : containing other components.
    3. parallel : also a composite, but re-dispatching calls to its external server interfaces towards its inner components.
  - It communicates with other components through 1-to-1 or group communications.
  - A primitive component is formed from one (or several) Active Object(s), executing on one (or several) JVM(s)
  - It provides a set of reflexive facilities (as defined in Fractal).
  - It implements several controllers for defining the non-functional aspects (i.e., binding controller allows to bind/unbind interfaces included in the components).

  A ProActive component composition can be specified by a descriptor, defined using the Fractal ADL (Architecture Description Language). Deployment of ProActive components relies on the notion of virtual node, capturing the deployment capacities and needs, and on XML deployment descriptors. ProActive components also allow to encapsulate legacy code.

5 Use-Case Scenarios

In this section, we include three different use-case scenarios to illustrate better and capture the practical requirements from a user's point of view.

5.1 Use-Case 1: GENIE: Grid ENabled Integrated Earth System Model

The GENIE (Grid ENabled Integrated Earth System Model) project [14] is an application which demonstrates the need for a scalable modular architecture. The project models the behaviour of large-scale thermohaline circulation, utilising various scientific modules corresponding to different environmental fragments. The case study would focus on componentizing the currently available serial solution for execution in a Grid platform. This task opens up a series of challenges including efficient componentization and composition, interface constraints, model-specific and resource-constrained simulations, real-time scheduling of operations, distributed execution and collection of large volumes of data.

This use-case would illustrate and cover a wide spectrum of questions pertaining to execution/adopting legacy applications to our generic, lightweight Grid platform and the capability of the platform in capturing and validating workflow models in scientific applications.

5.2 Use-Case 2: Visualisation of Large Scientific Datasets

This use-case, Visualisation of Large Scientific Datasets, captures the requirements for the platform to manipulate interactively and visualise large volumes of data in a Grid environment. The large datasets are partitioned offline but the operations for visualisation are determined at runtime using a front-end, such as the MayaVi tool [24]. Visualisation of a given dataset typically involves processing a "visualisation pipeline" of domain-specific data analysis and rendering components. This happens before rendering and includes operations such as feature extraction or data filtering computations. Osmond *et al.* [23] describe the implementation of a "domain-specific interpreter" that allows visualisation pipelines specified from MayaVi to be executed in a distributed-memory parallel environment.

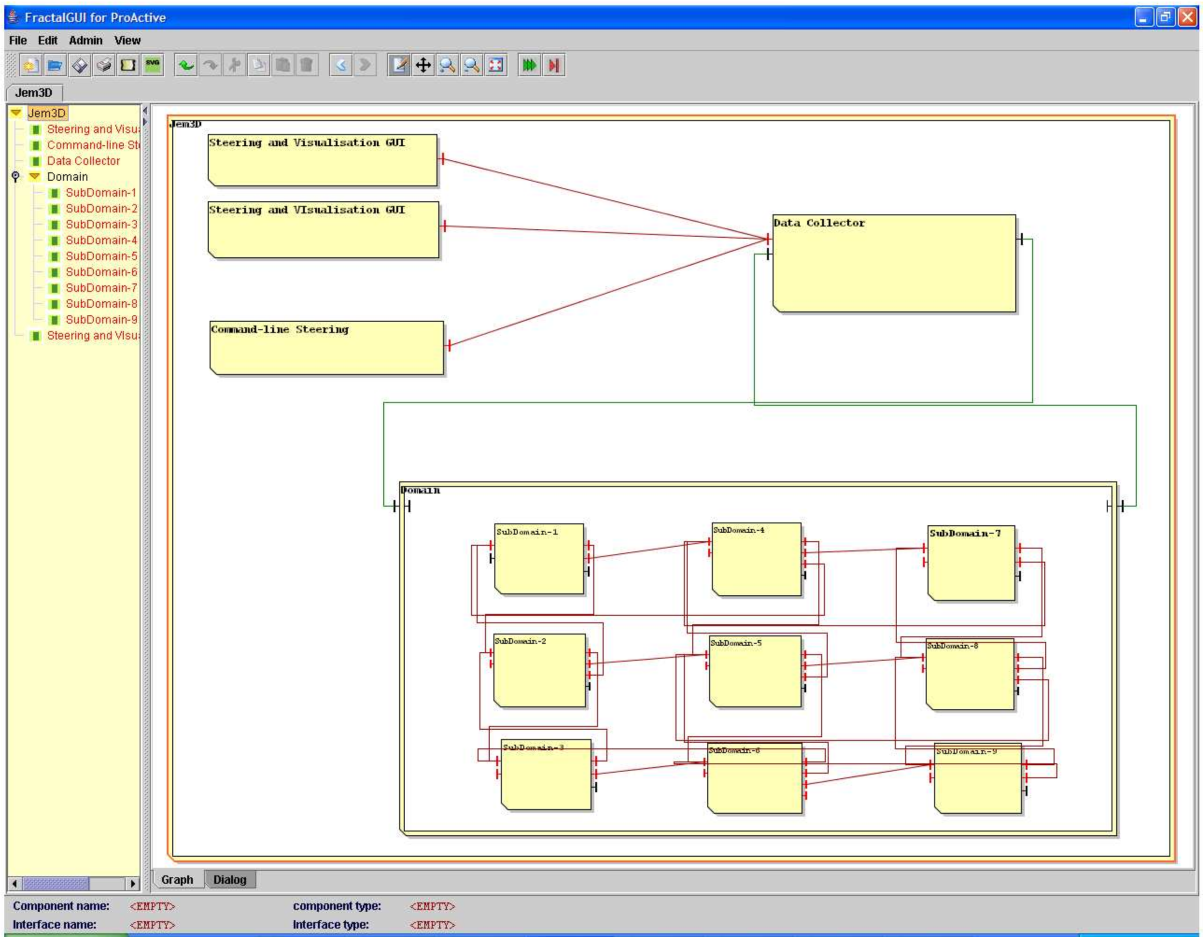
The key challenge posed by this use-case is a mechanism which can take such an execution plan (effectively a list of VTK operations to be performed) and execute it on Grid resources. In particular, this means

- A multi-language environment
- A lightweight mechanism for executing a script of Python operations on a remote Grid resource
- A lightweight mechanism for accessing the underlying datasets on remote resources (this could be done by file transfer, or — better, the resource mapping should take account of where the data is located)
- Ability to cache intermediate results on remote resources. This requirement can lead to significantly better performance when visualisations are repeated, and relies on some form of "remote state".

5.3 Use-Case 3: Jem3D: High Performance Numerical Solver

This use case concerns the development and deployment of high performance numerical solvers on the Grid. Specifically, we concentrate on the Jem3D application: a finite volume, time domain solver for the 3D Maxwell's equations modelling the propagation of electromagnetic waves [5]. Jem3D was developed at INRIA, and it has evolved from a Fortran/MPI parallel application to a grid-enabled, component-based application built on ProActive. Current implementation of the Jem3D relies on following components: steering and visualisation agents, data collectors, processing components (termed sub-domains) that correspond to a geometrical decomposition of the computational domain, and a composite component that encapsulates the processing components, as illustrated in Figure 1.

The Jem3D use case demonstrates several of the previously identified requirements. Dynamic component deployment is necessary for allowing users at separate workstations to participate at any time in monitoring and steering the application. Reconfiguration and adaptivity are essential for accommodating variations in the availability of underlying resources. In one scenario, the application adapts to performance degradation by dynamically changing the size of the computation. Alternatively, the application reacts by migrating an underperforming processing component to a more powerful node or by decomposing further its computational domain to sub-domains and assigning them to new, dynamically-deployed processing components. Another scenario concerns the data collector component that is



Component name: <EMPTY> component type: <EMPTY>
Interface name: <EMPTY> Interface type: <EMPTY>

Figure 1: Component Architecture in Jem3D

used to periodically receive computed solutions from processing components. If the load imposed on the collector machine becomes excessive, the application reconfigures itself to employ a hierarchical structure of collectors that exhibits better scalability. Enabling all previous scenarios requires dynamic metadata describing the deployment properties of components and the interconnections between them (e.g., interconnections between collectors and processing components). Finally, support for group communication is particularly useful for Jem3D because it greatly simplifies the implementation and allows it to adapt to changing application needs, such as the need for new solver algorithms. Moreover, group communication greatly improves the efficiency of the communications.

The above Jem3D requirements are supported by ProActive as follows. The platform enables dynamic component deployment on arbitrary machines using an extensible set of deployment protocols. In terms of reconfiguration and metadata, the platform supports maintaining and manipulating the interconnections between components, managing their lifecycle, and migrating them between arbitrary machines. Group communication is a key ProActive feature. The adaptivity scenarios are currently unsupported and form the subject of on-going work. Supporting adaptivity is expected to involve the introduction of manager components that build on the reconfiguration primitives already provided by the platform, without requiring changes to existing components.

6 Conclusions

In this paper, we have outlined our initial findings in designing a generic, lightweight Grid platform. With a component oriented methodology, we have proposed a set of requirements and features that a generic, lightweight Grid platform should support. We have paid special attention to ensuring that a wider class of applications and infrastructures are supported, including non-grid, legacy- and enterprise-class applications. We intend to achieve the required scalability by relying on dynamic, on-demand plugging of services and components. We have also captured user-centric views and requirements with the help of different use-cases.

Towards designing a platform, we would like to investigate the following issues:

- Dynamic non-interruptive reconfiguration of services/components
- Efficient life-cycle management of components
- Tools and supportive environments for using and porting non-Grid and legacy applications
- Realistic modelling and synthesis of Grid resources and components for deriving information to be used for providing adaptive, reconfigurable services.

References

[1] *ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing*, volume 3222. Springer-Verlag, 2004.

[2] I. Attali, D. Caromel, and A. Contes. Hierarchical and declarative security for grid applications. In *10th International Conference On High Performance Computing, HIPC*, volume 2913, pages 363–372. LNCS, 2003.

[3] Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, Joseacute, M. Cela, and Rogeli Grima. Programming Grid applications with GRID superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[4] L. Baduel, F. Baude, and D. Caromel. Efficient, flexible, and typed group communications in java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36. ACM Press, 2002.

[5] Laurent Baduel, Françoise Baude, Denis Caromel, Christian Delbé, Nicolas Gama, Said El Kasmi, and Stéphane Lanteri. A parallel object-oriented application for 3d electromagnetism. In *IPDPS*. IEEE Computer Society, 2004.

[6] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP2002)*, 2002.

[7] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, June 2002.

[8] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[9] Denis Caromel, Fabrice Huet, and Julien Vayssière. A simple security-aware mop for java. In *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 118–125, Kyoto, Japan, September 2001.

[10] CCA Forum Home Page. The Common Component Architecture Forum, 2004. http://www.cca-forum.org.

[11] Enterprise Grid Alliance. Reference model. Technical Report Version 1.0, Enterprise Grid Alliance, 2005.

[12] Matthieu Morel Francoise Baude, Denis Caromel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Italy*, volume 2888 of *LNCS*, pages 1226 – 1242. Springer, 2003.

[13] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.

[14] GENIE. The Grid ENabled Integrated Earth system model project. http://www.genie.ac.uk, 2005.

[15] Pawel Jurczyk, Maciej Golenia, Maciej Malawski, Dawid Kurzyniec, Marian Bubak, and Vaidy S. Sunderam. A system for distributed computing based on H2O and JXTA. In *Proceedings of the Cracow Grid Workshop, CGW'04, December 13–15, 2004*, pages 257–268, Kraków, Poland, 2005.

[16] Dawid Kurzyniec, Tomasz Wrzosek, Dominik Drzewiecki, and Vaidy Sunderam. Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters*, 13(2):273–290, 2003.

[17] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, and Aleksander Slomiński. RMIX: A multiprotocol RMI framework for java. In *Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 140–146, Nice, France, April 2003. IEEE Computer Society.

[18] William Lee, Anthony Mayer, and Steven Newhouse. ICENI: An Open Grid Service Architecture implemented with Jini. In *SC2002: From Terabytes to Insight. Proceedings of the IEEE ACM SC 2002 Conference*. IEEE Computer Society Press, 2002.

[19] Maciej Malawski, Dawid Kurzyniec, and Vaidy Sunderam. MOCCA – towards a distributed CCA framework for metacomputing. In *Proceedings of the 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2005)*, 2005.

[20] Anthony Mayer, Andrew Stephen McGough, Nathalie Furmento amd Jeremy Cohen, Murtaza Gulamalim, Laurie Young, Ali Afzal, Steven Newhouse, and John Darlington. *Component Models and Systems for Grid Applications*, chapter ICENI: An Intergrated Grid Middleware to Support e-Science, pages 109–124. Springer Verlag, 2004.

[21] Andrew Stephen McGough, William Lee, and John Darlington. ICENI II Architecture. In *UK e-Science All-Hands Meeting*, September 2005.

[22] Object Management Group, Inc. CORBA, 2005. http://www.corba.org/.

[23] K. Osmond, O Beckmann, A.J. Field, and P.H.J. Kelly. A domain-specific interpreter for parallelizing a large mixed-language visualisation application. To Appear in Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing.

[24] Prabhu Ramachandran. MayaVi: A free tool for CFD data visualization. In *4th Annual CFD Symposium, Aeronautical Society of India*, August 2001. mayavi.sourceforge.net.

[25] Rob van Nieuwpoort and Jason Maassen and Gosia Wrzesinska and Rutger F. H. Hofman and Ceriel J. H. Jacobs and Thilo Kielmann and Henri E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency - Practice and Experience*, 17(7-8):1079–1107, 2005.

[26] S.Kohn, G. Kumfert, J. Painter, and C.Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *Proc. of the 10th SIAM Conf. on Parallel Processing for Sci. Comp.*, Portsmouth, USA, March 2001. SIAM.

[27] Jeyarajan Thiyagalingam, Stavros Isaiadis, and Vladimir Getov. Towards Building a Generic Grid Services Platform: A Component Oriented Approach. In Vladimir Getov and Thilo Kielmann, editors, *Component Models and Systems for Grid Applications*, pages 39–46. Springer, 2005.