Proceedings of the CoreGRID Workshop on Grid Systems, Tools and Environments, 1st December 2006, Sophia-Antipolis, France.

Editors:

Rosa M. Badia[1]
Françoise Baude[2]
Vladimir Getov[3]
Thilo Kielmann[4]
Ian Taylor[5]

[1] Universitat Politecnica Catalunya, Barcelona, Spain
[2] INRIA-I3S CNRS-UNSA, Sophia Antipolis, France
[3] Harrow School of Computer Science, University of Westminster
[4] Vrije Universiteit, Amsterdam, The Netherlands
[5] Cardiff University, UK

This is a reproduction of CoreGRID Technical Report Number TR-0103, 14 September 2007 and is reprinted here with permission.

The report is available on the CoreGRID website, at:

http://www.coregrid.net/mambo/images/stories/TechnicalReports/tr-0103.pdf

CoreGRID Workshop on Grid Systems, Tools and Environments, 1st December 2006, Sophia-Antipolis, France

Proceedings

Editors: *Rosa M. Badia, Françoise Baude, Vladimir Getov, Thilo Kielmann, Ian Taylor*

CoreGRID Technical Report
Number TR-0103
14th September 2007

Institute on Grid Systems, Tools & 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

Preface

The CoreGRID institute on Grid Systems, Tools, and Environments (STE) has held a workshop on the state of the art in its area of expertise during the Grids@Work'06 event in Sophia-Antipolis, on 1 December 2006. In this report, the workshop organizers present the proceedings volume of the workshop, containing the presented papers.
The workshop aim was to bring together presentations describing both state of the art and ongoing work in the area of Systems, Tools and Environments for Grid Applications. According to the objectives of the STE Institute, submissions on the following (non-exclusive) list of topics were expected:

- Generic, component-based Grid platforms
- Mediator components, integrating applications and systems
- Integrated toolkit approaches
- Advanced tools and environments for problem solving
- Use of GCM for Grid system software design
- Interoperability between peer-to-peer and client-server Grids

The expected focus was on hands-on experience with building component-based Grid environments, and the integration of applications with system software.
Submissions authored by multiple CoreGRID partner institutions were highly welcomed and such submissions were given preference over single-institution papers in the selection process.
The contributions presented at the workshop where selected by peer-review of abstract submissions. From the submission, nine contributions were selected. Authors were asked to submit a whole paper of the contribution, which are included in this techical report.
The workshop was organized in four sessions. The first sessioni, after the welcome and introductions, was basically devoted to the presentation of the keynote speaker, Alessandro Bassi (HITACHI Data Systems Corporation), with the presentation of *Data Management Challenges in Dynamic Enterprise Grid Systems*.
Second session was focused on *Components-Based Design Methodologies* and besides five presentations of contributions sent to the workshop, a presentation on the status of deliverable D.STE.05 was given. Third session was focused on *Integrated Environments* with four more presentations of contributions sent to the workshop. Finally, last session was invested in the discussion of the former contributions and in the definition of future plans and directions of the institute.
The workshop organizers would like to thank all contributors to the workshop without whom this event would not have been possible. We appreciate the technical papers from the STE institute, and especially the keynote speaker who provided us with valuable insights from his own perspective. Last but not least, we would like to thank the local teams from Sophia-Antipolis (both INRIA and ETSI) who have provided us with an excellent meeting infrastructure.

Index

Workshop program

11.00 - 12.00	Opening Session Session Chair: Vladimir Getov
11.00 - 11.30	Welcome Message and Overview, Vladimir Getov, WP7 CoreGRID Leader
11.30 - 12.00	Invited talk Data Management Challenges in Dynamic Enterprise Grid Systems Vincent Franceschini and Alessandro Bassi, HITACHI Data Systems Corporation
12.00 - 13.20	Lunch Break
13.20 - 15.00	Session I - Components-Based Design Methodologies Session Chair: Rosa M. Badia
13.20 - 13.40	Task 7.3: Integrated Toolkit Deliverables R. Badia, E. Tejedor
13.40 - 14.00	Hierarchical MPI-like Programming using GCM Components as Implementation Support F. Baude, D. Caromel, N. Maillard, E. Mathias
14.00 - 14.20	A Case Study of Building Mediator Components with the Fractal Model M. Ejdys, U. Herman-Izycka, T. Kielmann, V. Getov, R.M. Badia, I. Taylor
14.20 - 14.40	Designing a Reconfigurable and Extensible Lightweight Grid Platform: Application to Deployment Management J. Thiyagalingam, N. Parlavantzas, S. Isaiadis, L. Henrio, V. Getov
14.40 - 15.00	An ADL-based Support for CCA Components on the Grid M. Malawski, T. Bartynski, E. Ciepiela, J. Kocot, P. Pelczar, M. Bubak
15.00 - 15.20	Communication Models for Processes and Services in Mobile Lightweight Grids V. Georgiev, S. Isaiadis, V. Getov, L. Kirchev
15.20 - 15.40	Coffee Break
15.40 - 17.00	Session II - Integrated Environments Session Chair: Thilo Kielmann
15.40 - 16.00	Executing Parameter Study Workflows in the P-GRADE Portal G. Sipos, A. Goyeneche, T. Kiss, P. Kacsuk
16.00 - 16.20	Legacy Code Repository with Broker-based Job Execution G. Kecskemeti, G. Terstyanszky, T. Kiss, P. Kacsuk
16.20 - 16.40	Information Models for Lightweight Grid Platforms G. Pashov, K. Kaloyanova, K. Boyanov
16.40 - 17.00	Multiple Broker Support by Grid Portals A. Kertsz, Z. Farkas, P. Kacsuk, T. Kiss
17.00 - 18.00	Session III - Discussion on Future Plans and Directions Session Chair: Vladimir Getov

Session I: Components-Based Design Methodologies

Hierarchical MPI-like Programming using GCM Components as Implementation Support

Françoise Baude*, Denis Caromel*, Nicolas Maillard+ and Elton Mathias *+

{Françoise.Baude, Denis.Caromel}@inria.fr
nicolas@inf.ufrgs.br, Elton.Mathias@inria.fr

*INRIA-I3S CNRS-UNSA
Sophia Antipolis, France

+UFRGS - Universidade Federal do Rio Grande do Sul
Porto Alegre/RS - Brazil

Abstract

While MPI became a standard on programming model for developing cluster applications, so far none of the many programming models proposed for Grids have reached such status. For this reason, using MPI in Grids is being considered a good cost-effective solution. Different from other related works that mainly intend to ease the execution of MPI processes in Grids, we understand that for obtaining good performance, it is mandatory to adapt algorithms and applications to reflect the Grid topology. For this reason, we introduce some extensions to the MPI standard that include new communicators for the communication among different clusters and some primitives addressing topology discovery. The support to these features is offered by an underlying component model based on the GCM and the ProActive/GCM implementation. Through a hierarchy of components and their collective interfaces, the main MPI original primitives could be used to easily develop and run MPI applications in Grid environments.

1 Introduction

Several programming models have been proposed for Grid programming. Nonetheless, so far, none of them met all the requirements, namely dynamicity, scalability and performance. Differently, in the field of high performance computing in clusters, the message passing model became a standard with a large number of available libraries and legacy applications. For this reason, the idea of using the well known and accepted MPI in Grids is considered a good cost-effective solution.

While not a high level programming model by any means, the message passing model lacks dynamicity and abstractions to program Grid applications. Indeed, the MPI standard addresses cluster environments, not having primitives adapted to program Grid environments, that are inherently hierarchical. Contrary to message-passing, a component-based model encompasses most of the proposed Grid programming models (message passing, distributed objects, skeleton-based programming, service-oriented and workflow models) as it provides most of the features presented by other models and in addition, the capability of encapsulating code. Thus, it should be more adequate to program Grid applications.

This work intends to propose an hybrid model that combines the high performance and high acceptability of the MPI standard with the flexibility of the component-based programming model in order to meet Grid programming requirements, offering MPI programmers a way to develop their applications/algorithms in a Grid-aware hierarchical manner, yet taking profit of legacy high-performance codes.

Our solution will rely on the addition of new MPI communicators that models the hierarchical structure of Grids and a related API, that may offer an abstraction well-suited to programmers used to MPI in order to reflect the hierarchical topology of the Grid within the deployed application. Indeed, Grid related features will be considered in the implementation of these primitives, in a transparent way for MPI programmers. For this, the implementation will be based on the ProActive platform, which also offers a pre-prototype implementation of the CoreGRID GCM component model.

This paper presents the outline of the work, that is currently on its definition phase. In the next section, we present background information. Then, in the section 3 we present the proposition of the work, that includes the general principles and specification. Some initial implementation issues are presented in the section 4. Section 5 concludes this paper.

2 Background

2.1 The ProActive Middleware

The ProActive middleware [1] is a 100% Java middleware, which aims to achieve seamless programming for concurrent, parallel, distributed and mobile computing. As it is built on top of standard Java API, it does not require any modification of the standard Java execution environment, nor does it make use of a special compiler, pre-processor or modified virtual machine.

The base model is an uniform active object programming model. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls. Active objects are remotely accessible via asynchronous method invocation. This is provided by automatic future objects as a result of remote methods calls, and synchronization is handled by a mechanism known as wait-by-necessity. Besides of a programming model, ProActive features: group communication with dynamic group management, an object oriented SPMD model (OOSPMD), code mobility, fault tolerance with check-pointing, a powerful deployment model based on XML descriptors that offers support to numerous network protocols, cluster resource managers and Grid tools as well as a component programming model based on the Fractal specification.

Specially interesting to this work is the pre-prototype of the the ProActive/GCM (Grid Component Model) and the ProActive MPI Code Wrapping Mechanism. The next two subsections present them in more details.

2.2 The ProActive/GCM (Grid Component Model)

The Grid Component Model (GCM), defined by the Institute on programming models of the CoreGRID project, defines a lightweight component model (the GCM) for the design, implementation and execution of Grid applications. The key problematics addressed by the GCM are programmability, interoperability, code reuse and efficiency.

This model relies on the Fractal component model as a basis for its specification. In fact, GCM can be considered an extension to the Fractal specification, addressing Grid requirements like deployment and collective communication. In this sense, the ProActive/GCM [6] is a reference implementation of the GCM which provides a component framework that aims at fulfilling the needs of Grid programming.

The ProActive/GCM implementation introduced, as extensions to the Fractal component model, the definition of collective interfaces: multicast, gathercast and an optimized way to combine a gathercast plus a multicast interfaces for binding two hierarchical components composed of a different number of similar components, yielding to a MxN collective interface. These collective compositions enable the externalization, at the component level, of its parallel nature. And, as a consequence, the built-in possibility of collective synchronization and communication operations.

2.3 The ProActive MPI Code Wrapping Mechanism

The Code Wrapping Mechanism, recently introduced in the ProActive middleware proposes a simple wrapping method designed to automatically deploy MPI applications on clusters or desktop Grids. Besides, a communication API enables the coupling of several codes, MPI and/or Java.

Two kinds of applications may be interested by the code wrapping mechanism: first, almost unmodified legacy MPI applications coupling and second, the development of conventional stand-alone Java applications using pieces of MPI legacy codes.

The main features of the wrapping mechanism includes:

- Transparent wrapping and deployment of MPI applications: consists in wrapping MPI processes within ProActive active objects, adding capabilities for deployment, control and communication. Due to the use of the ProActive deployment descriptors, an MPI application can be deployed using a large number of protocols and schedulers. In addition, the ProActive file transfer mechanism enables the transfer of application binaries and input data.

- Support for "MPI to/from Java" point-to-point communication: a C library, a set of MPI-like C functions and Java classes permit the exchange of messages between the two worlds. Furthermore, this feature adds capability of communication between two MPI process potentially located at different domains under firewalls or even private IPs.

- Control of MPI processes: this makes possible externally trigger job execution, kill jobs, synchronize them and retrieve execution status and results.

The main intention of the code wrapping mechanism is the transparent execution of MPI applications in Grids adding the possibility of coupling two or more MPI codes together in a non-SPMD fashion. However, in order to use most of the features (mainly the most interesting ones), the programmer must call the new functions from within the MPI codes, and also write some pieces of ProActive code for control and deployment of the wrapped code.

3 Proposition

3.1 General Principles

We consider in this work that Grids are hierarchical by nature: i.e. a Grid can be considered as a set of multi-core nodes regrouped on multi-processors PCs, organized within clusters, then interconnected through wide-area networks. Consequently, one of the challenges is to be able to take into consideration the physical topology of the Grid within the application.

On a hierarchical scenario, if a communication occurs between two processes running onto the same cluster, it is perfectly reasonable that one should communicate using a standard plain MPI communication, usually implemented in an efficient manner. Whereas if the two processes run onto different clusters, the communication may need to cross firewalls, to be cyphered, etc., at least a higher latency will be incurred due to the longer distance the message needs to travel.

From the programmer point of view, taking into consideration such constraints at the application level requires at least a new organization of the parallel algorithm: to give preference to communications between

MPI processes that are neighbours in the cluster instead of communications between MPI processes lying onto different clusters of the Grid.

As briefly presented in the section 2.3, the initial ProActive wrapping mechanism intends to couple two or more MPI applications together. For this purpose, an additional API is offered to programmers if they want to design their MPI application for subsequent coupling with one or several other (possibly MPI-based) applications. On the contrary, we aim here to propose an extension to the message passing programming model adding the abstraction of hierarchical communicators, applicable within a single SPMD application. The consequence is that the programmer should agree to design its SPMD applications in a hierarchical manner, by clearly organizing the code and algorithm so that all the communications between MPI process are clearly arranged depending on the multi-level Grid hierarchy. The hierarchical approach will take advantage of the notion of MPI communicators in a hierarchical fashion, so that the introduced API fits more closely the MPI style, i.e. through the abstraction of ranks and communicators.

Unlike the usual MPI environment, Grids are also dynamic by nature. This characteristic leads to the fact that the allocated resources are only known at runtime, requiring the communicators management to be done directly by the framework. Besides of the communication process, we define an API for topology discovery, e.g. number of neighbours each MPI process has at each level in the hierarchy, connectivity, etc. Indeed, we aim to define a flexible and non error-prone way for the programmer to identify the MPI process(es) with which to communicate.

3.2 Specification

3.2.1 Extensions to the Message Passing Interface

As previously referred in the section 3.1, we aspire to keep as much as possible the MPI programming style. In order to do so, we provide some clear and intuitive extensions to the MPI, that include support for hierarchical communication and topology discovery. Through the use of this extended interface, programmers will be capable of transparently deploy and execute their MPI applications, having the Grid complexities hidden by the framework.

This extensions include new MPI communicators and some new primitives.

New MPI communicators Ideally, we consider a Grid (Level 1 of the hierarchy, or just L1 for simplicity) as a five levels architecture:

- the second level (L2) consists of a small number (< 10) of geographical sites (Grid nodes). These Grid nodes are interconnected by a wide area network (WAN), and so, we can expect majors delays and limited bandwidth on the connection;
- the third level (L3) consists of the set of clusters defining a Grid node. These clusters are typically interconnected by 1 Gbit/s links.
- the fourth level (L4) consists of the set of processing nodes defining a cluster. On each Grid node, the aggregated number of processing nodes is between a few hundreds to a thousand nodes, usually with a fast network;
- the fifth and lowest level allows to characterize the architecture of a node: single versus multiple-processor systems, single versus multiple-core systems and memory structure (SMP, NUMA, etc.). This level is treated by the operating system and for this reason is out of the scope of this work.

As standard MPI already addresses the fourth level and the operating system the fifth, we introduce four new communicators to deal with the first, second and third levels:

- `MPI_COMM_SITE`: communicator containing all the allocated nodes within a given site in the Grid.
- `MPI_COMM_GRID` : communicator containing all the allocated nodes in the Grid.
- `MPI_COMM_SITE_GATEWAYS`: this is a special communicator that aggregates one node (usually the rank 0) of each cluster within a site. The intention of this communicator is to offer an abstraction to induce an explicit hierarchical programming, and can be used in order to reduce the many indirections (and performance degradation) of global communications.
- `MPI_COMM_GRID_GATEWAYS`: the same as the previous one, but in the higher level. Usually, the node with the rank 0 of the first cluster of a given site.

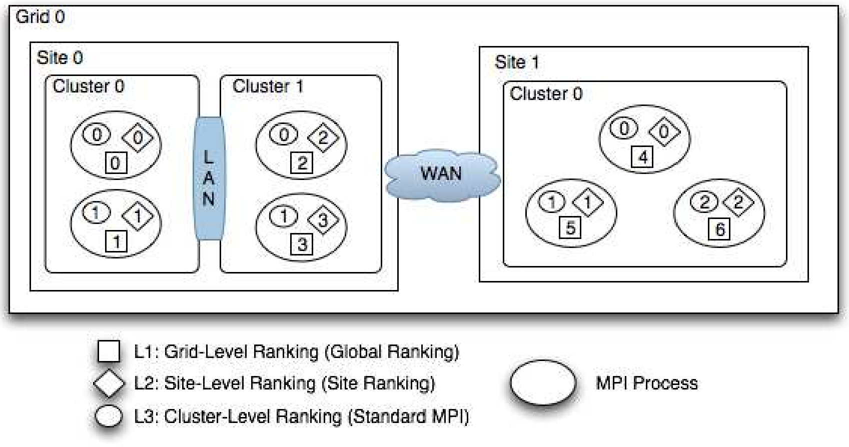

Grid 0
Site 0
Cluster 0
Cluster 1
LAN
WAN
Site 1
Cluster 0
L1: Grid-Level Ranking (Global Ranking)
L2: Site-Level Ranking (Site Ranking)
L3: Cluster-Level Ranking (Standard MPI)
MPI Process

Figure 1: Hierarchical Communicators and ranks

The figure 1 shows how the communicators and ranks are organized. This is a pretty simple yet powerful way of giving ranks to MPI processes in a hierarchical environment. Simple because it could be done during the startup of the environment, so having an unified global view. Powerful as it enables an easy and non-error prone way to identify processes, indeed keeping the communicator abstraction.

New primitives In addition to the abstraction of hierarchical communicators, we include a set of primitives, mainly addressing topology discovery. Some of the most useful are:

- `MPI_Comm_gatewayRank (MPI_Comm comm, int *gateway_rank)`: Different from other new communicators, it's possible that a given rank doesn't figure in the `*GATEWAYS` communicators. In this case, it's useful to know an entry point process in order to make a hierarchical communication;
- `MPI_Comm_gatewaySize (MPI_Comm comm, int gateway, int *size)`: Retrieve the number of process existent on a cluster (or site), that are associated to a gateway;
- `MPI_Comm_translate(int in_rank, MPI_Comm, int* out_rank, MPI_Comm ):` translate ranks between communicators.

Some other useful primitives are functional with the new communicators, such as `MPI_Comm_rank` and `MPI_Comm_size` . However, at least in the first prototype we don't intend to provide full support to all the already existing primitives, when used with the new communicators. In fact, just a minimum set of the most used original primitives will be supported.

3.2.2 The Underlying Component Model

The primitives presented above and all the communication mechanisms that includes the new communicators are treated by a component that wraps the caller process. This component is the base element (GCM primitive component) which forms the underlying component model that supports the added feature. For performance purposes, direct and collective communications within a single cluster, instead of relying of bindings between primitive GCM components, will be done through direct MPI communications.

All primitive components that are neighbours in the lower level (L4) are embedded into a composite component that represents the cluster (L3 in the hierarchy). These composite GCM components are interconnected within a site (L2), because their inner components (MPI processes) contain data subject to further sharing, so subject to intercommunications. For this relationship to be translated into component bindings, the GCM-based runtime support will rely on the information obtained from the data partitioning tool. The same applies regarding the top level that aggregates all the sites (L1).

The primitive component L4 (Figure 2 a.), that wraps the MPI processes presents 2 client interfaces (L3C and L2C, meaning Level 3 client and server, respectively) and 2 server interfaces (L3S and L2S): L3C is bound to the enclosing composite L3 and carries messages through it to other components of L4 from the same site; L2C is also bound to the composite L3 and carries message through L3 and L2 components to L4 components running onto different sites. As previously mentioned, all the communication among processes within the same cluster are treated at MPI level. The server interfaces correspond directly to these client interfaces and are responsible for receiving messages from the other side.

The composite component L3 (Figure 2 b.) represents a cluster and presents 4 client interfaces (L3C, L2C, gcastL3C and gcastL2C) and 3 server interfaces (called L3S, L2S and gcastL2S). The L3C permits to send messages to other cluster(s) of the same site, while the L2C tunnels messages to cluster(s) present in other site(s) via L2 composites. The collective interfaces gcastL3C and gcastL2C are bound to the other L3 components within the same site and to the corresponding L2 composite and can be configured to perform, at component level, collective MPI operations such as data MPI_bcast and MPI_scatter within a given site or multiple sites. The L3S and L2S server interfaces are responsible to carry messages from the primitives to the others clusters within a site or cluster of other sites (through the L2C to L2 composites). Also, these interfaces (L3S and L2S) are useful to perform synchronization operations. The gcastL2S is responsible for collective operations such as MPI_gather and barriers.

The composite component L2 (Figure 2 c.) represents a site and includes 2 client interfaces (L2C and gcastL2C) and 2 server interfaces (L2S and gcastL2S). These interfaces are responsible for the exchange of data among different sites within a Grid, so enabling the different MPI group communication patterns in a Grid level. The L2C used in most of the operations including point to point communication between process within different sites or broadcast operations while the gcastL2C is useful for transmitting data in scatter operations. The server interface L2S is used to tunnel messages from inner components to outside of a site as well as synchronization operations. The gcastL2S server interface responsible for receiving messages from other sites but in gather operations.

The composite component L1 (Figure 2 d.) represents a Grid and it is just an abstraction as we do not have a higher-level abstraction to gather different Grids.

Basically the communication from one MPI process within a cluster to an other one which is hosted in an other cluster, should normally go through the hierarchy of GCM composite. Even if it will be probably less efficient than an intra-cluster communication, it gives the opportunity to establish inter-cluster communications that should not be possible otherwise, eg. due to firewalls, private IP, etc. This is assumed to be the standard case. Nevertheless, for optimization purpose, whenever possible a direct binding between two GCM components not belonging to the same composite(s) will be established. The section 3.3.2 discusses in more details this enhancement.

The figure 3 shows the hierarchy of components, corresponding to the rank mapping of the figure 1. The

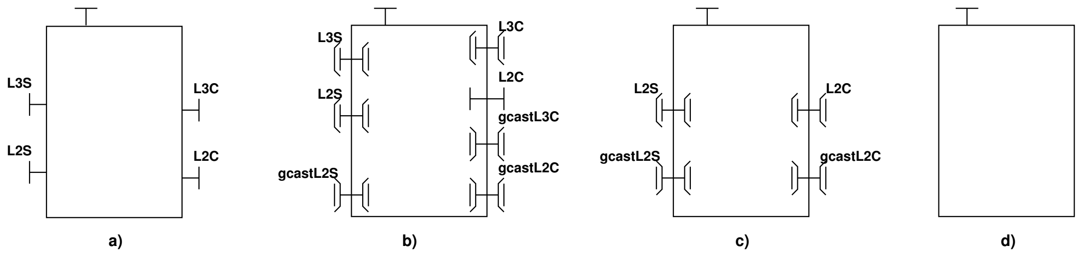


Figure 2: a. Primitive L4; b. Composite L3 (cluster); c. Composite L2 (Site); d. Composite L1(Grid)

arrows represent the exchange of messages necessary for a broadcast from one MPI process to the entire Grid. The communication within the same cluster is done directly through MPI native communication while the communication between different clusters on the same site passes by the enclosing components that represent the cluster, and the communication between different clusters of different sites passes also through the components that represent the sites.

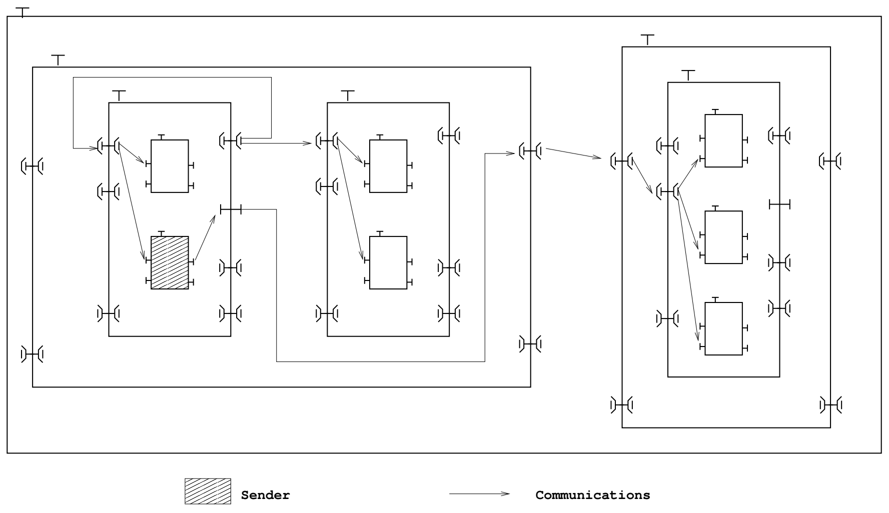


Figure 3: Global Broadcast throughout the Grid

3.3 Some Initial Implementation Issues

3.3.1 Improving communication between MPI processes and components through Java Native IO

In general, a point that is expected to cause severe impact in performance in any platform is the interchangeable usage of different technologies (in our case, languages). Currently, within the ProActive Code-Wrapping mechanism, "inter-system" message passing is implemented by ProActive asynchronous method invocation and the "intra-system" communication through the use of JNI (Java Native Interface) layered on top of IPC system V.

Besides of the need of extra copies (from native memory to the operating system message queue and then to inside of the JVM stack or vice-versa), this mechanism generates a bottleneck in the operating system queue. Also, it is not a portable solution.

Since the 1.4 release of the J2SE, Java offers a mechanism that enables direct access to native memory called Java NIO [3] through the package java.nio. A new implementation of the communication would further improve performance as it would avoid bottlenecks in the operating system and intermediary copies (so we could expect a performance quite comparable to the MPI native communication), and keep portability, natural in the Java world and in the MPI to some extend.

3.3.2 Enabling shortcuts to cross-cut components edges

One of the main drawbacks of using the component model is that communication between components in a hierarchical model, in our case an MPI communication tunneled through components, may involve crossing several membranes of enclosing composite components, and therefore paying the cost of several indirections. This tends to be even worse in the proposed model as the membrane is usually located remotely. If the invocations are not intercepted in the membranes, then it is possible to optimize the communication path by shortcutting: communicating directly from a caller component to a callee component by avoiding indirections in the membranes.

In the case of MPI applications, one process may decide to communicate with any other process and they may be potentially located onto different clusters. In this sense, for a highly optimized communication, we should have in advance configured a shortcut from all process to all process. However, having all-to-all shortcuts would incur a large overhead in setup mainly.

We tackle this problem by requiring from the associated deployment tool that it configures bindings between primitive components domain decomposition partition. Also, we assume that a process which communicates directly with another process is more likely to do it again.

Considering these suppositions, the shortcut mechanism offered by ProActive/GCM seems to suits well as it relies on a tensioning technique [6]: the first invocation determines and defines what is the shortcut path, then the following invocations will use this shortcut path.

It is obvious that nothing can be done if we are in a firewalled environment or communicating process in machines under private IPs. However, when possible, this mechanism tends to greatly improve performance, by reducing the indirections at the component level and also bottlenecks at the membranes.

The Figure 4 shows a standard communication (dashed line) and a shortcut (solid line) between two MPI processes in different sites of the Grid.

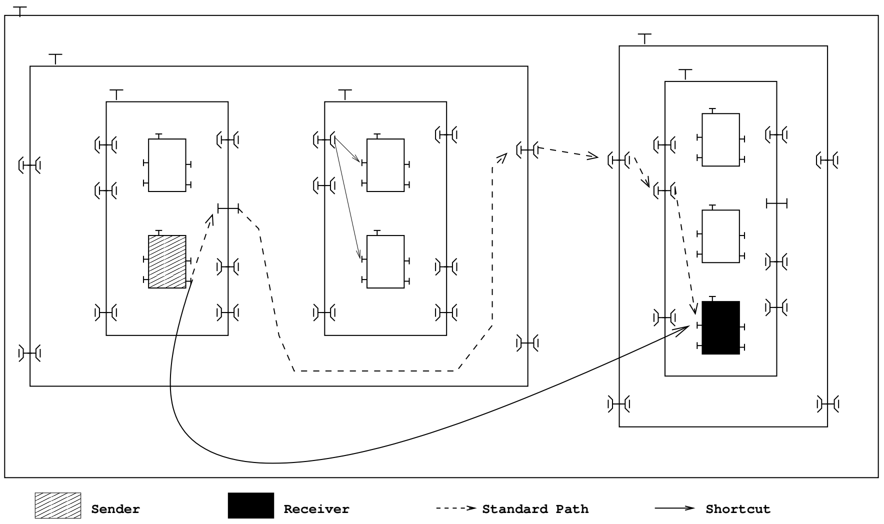


Figure 4: Shortcut in communication between processes in different sites

4 Related Works

Many of the reasons that motivate this work also motivated several research projects, clearly the spread usage of the MPI standard, clear interface and high performance of its implementations. Even with completely different approaches, the related works intend to address Grid issues in order to make MPI an interface to develop Grid applications.

This is the case for instance of MPICH-G2 [4], which uses Globus for authentication, authorization, executable staging, process creation, process monitoring, process control, communication, redirection of standard input and output and remote file access. PACX-MPI [2] is another implementation of MPI geared towards Grid computing, which contains a wide range of optimizations for Grid environments, notably for efficient collective communications. MagPie [5] also focuses on the efficiency of collective communications: it can optimize a given MPI implementation by replacing the collective communication calls with optimized routines.

Different from these works, we intend to provide not just an easy way to execute MPI applications in Grids but also an extension to MPI that offers support to develop grid-aware applications that take profit of the hierarchical architecture of Grids. We understand that it is mandatory to adapt algorithms developed to run in cluster in order to obtain a good performance in Grids, usually because differences between intra and inter-cluster communications are not negligible. However, this issue has been poorly treated in research so far.

5 Conclusion

This paper presented the outline of the work that intends to develop an hybrid model that combines the well accepted MPI standard with the flexibility of the component-based programming in order to meet Grid requirements, namely performance, heterogeneity and dynamicity.

The solution presented provides some extensions to the Message Passing Interface standard so that programmers used to this interface could easily adapt their application to a Grid environment and its hierarchical structure. In order to support these extensions, we propose the use of the ProActive middleware that already presents a code wrapping mechanism and several features that address Grid related issues. Worthy to note, the recently introduced ProActive/GCM implementation offers a very flexible and powerful way to deal with group communication and the many MPI flavours to do so.

We understand that this approach may present some open issues when compared with pure native implementations, mainly regarding performance. Nonetheless, we expect feasible solutions to tackle these questions. The general objective is ambitious, as it aims at providing a new grid-aware programming model: not a radically new one, but on the contrary, one that could have a chance to be adopted in the HPC community, i.e. by programmers used to MPI. Indeed, as a first framework built on top of GCM components, it can be considered a good practical evaluation of the GCM.

References

[1] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[2] Thomas Beisel, Edgar Gabriel, and Michael Resch. An extension to MPI for distributed computing on MPPs. In *PVM/MPI*, pages 75–82, 1997.

[3] Ron Hitchens. *Java NIO*. O' Reilly &Associates, Inc., 2002.

[4] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, may 2003.

[5] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[6] Matthieu Morel. *Components for Grid Computing*. PhD thesis, Universite de Nice - Sophia Antipolis, November 2006.

A case study of building mediator components with the Fractal model

M. Ejdys, U. Herman-Iżycka, T. Kielmann
kielmann@cs.vu.nl
Vrije Universiteit, Amsterdam, The Netherlands

Vladimir Getov
V.S.Getov@westminster.ac.uk
University of Westminster, UK

Rosa M. Badia
rosab@ac.upc.edu
Universitat Politecnica Catalunya, Barcelona, Spain

Ian Taylor
Ian.J.Taylor@cs.cardiff.ac.uk
Cardiff University, UK

Abstract

1 Introduction

Components allow for separation of concerns when designing a system. The Fractal component specification, which is the basis for this report, enables several cases of separation. Most importantly, Fractal allows to separate functional and configuration aspects. Our main goal was to investigate how the Mediator Toolkit could benefit from using a component framework such as Fractal.

This report is structured as follows. First, we describe the Fractal specification in Section 2. Additionally, we present Julia, its reference implementation as well as two other interesting implementations: aspect-oriented AOKell and Grid-oriented ProActive. In Section 3, we show how the Fractal framework could be used in the Mediator Toolkit.

2 Fractal

Fractal is a modular and extensible component model that can be used with various programming languages to design, implement, deploy and reconfigure various systems and applications, from operating systems to middleware platforms and to graphical user interfaces.

The model heavily relies on the *separation of concerns* design principle, that is; various concerns or aspects of an application should be separated into distinct pieces of code or runtime entities. In particular, Fractal uses three specific cases of this principle: separation of interface and implementation, component-oriented programming, and inversion of control (components are configured and deployed by an external, separated entity).

2.1 Fractal Specification

This section presents an overview of the Fractal specification [1].

The main goals of the Fractal component model are to implement, deploy and manage (i.e. monitor and dynamically reconfigure) complex software systems. These goals motivate the main features of the Fractal model:

- composite components (to have a uniform view of applications at various abstraction levels)
- shared components (to model resources)
- introspection capabilities (to monitor a running system)
- configuration and reconfiguration capabilities (to deploy and dynamically reconfigure an application)

Therefore, the Fractal specification can be thought of as an extensible system of relations between well defined concepts. However, components do not have to implement all of these concepts. Rather, they can provide a well defined subsets of specifications, which we present here with respect to the increasing "level of control".

2.1.1 Control Level 0: basic

The component at this level is a runtime entity that does not provide any control capability to other components. This is similar to an object in that it can only be used in one way, namely by calling its methods.

2.1.2 Control Level 1: external view

At external "introspection" level component provides a standard interface that allows the discovery of its external interfaces.

Before we proceed note that a component interface is an access point to a component that implements a language interface. Therefore component interfaces and language interfaces should not be mixed up. Importantly, a component interface is an access point that implements a language interface, whereas a language interface is a type. Despite of this risk, sentences such as "a component has an interface that implements the language interface "will often be abbreviated, into "a component has an interface X", in order to improve readability.

There are two kinds of interfaces: a client (or required) interface emits operation invocations, while a server (or provided) interface receives them. This is illustrated in Figure 1.

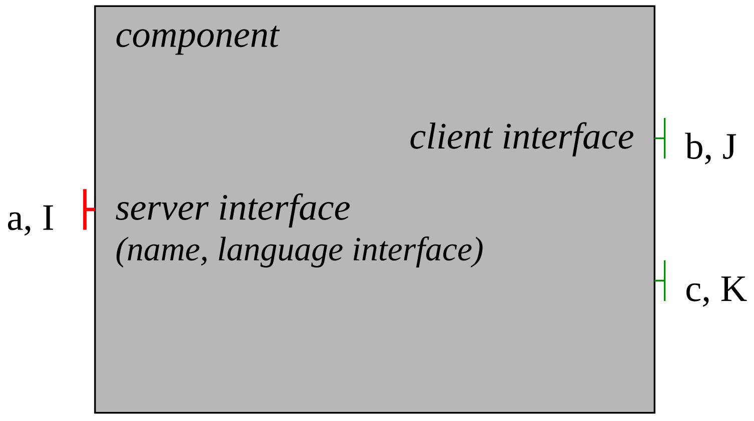


Figure 1: External view of the Fractal component.

In order to discover the external interfaces of a component, a component can provide an interface that implements the special Fractal *Component* interface. This language interface provides two operations named `getFcInterfaces` and `getFcInterface`, which can be used to retrieve the interfaces of the component.

2.1.3 Control Level 2: internal view

At the "configuration" level, a component provides control interfaces to introspect and modify its content (i.e. sub-components).

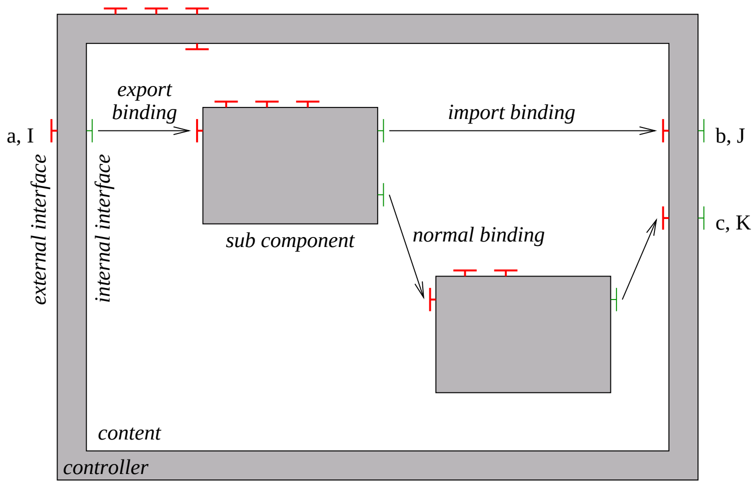


Figure 2: Internal view of the Fractal component.

Internally, a Fractal component consists of two parts: a *controller* (also called membrane); and a *content* (see Figure 2). The content of a component is composed of (a finite number of) other sub-components, which can be recursively nested. There are different types of components: composite (expose their content); primitive (do not expose their content; but have at least one control interface); and base (without control interfaces).

The controller of a component embodies the control behavior associated with a particular component, such as: providing the component's content, intercepting oncoming and outgoing operation invocations; and superposing control over its content (suspending, check pointing and resuming of sub components).

The controller of a component can have *external* and *internal* interfaces. External interfaces are accessible from outside the component, while internal interfaces are accessible only from the component's sub-components. A *functional* interface is an interface that corresponds to a provided or required functionality of a component, while a *control* interface is a server interface that corresponds to a "non functional aspect", such as introspection, configuration or reconfiguration (where interface name equals `component`, or ends with `-controller`).

Binding A binding is a communication path between component interfaces. The Fractal model distinguishes between *primitive* bindings (client-server interfaces bound) and *composite* bindings (arbitrary number of component interfaces of arbitrary language type bound).

Controller The supplied component controllers in the Fractal model include, the:

- *AttributeController* interface (to read and write attributes from outside the component)
- *BindingController* interface (to bind and unbind client interfaces to other components through primitive bindings). This interface defines the following operations: `listFc`, `lookupFc`, `bindFc`, and `unbindFc`.
- *ContentController* interface (to add and remove sub-components). This interface defines the following operations: `getFcInternalInterfaces`, `getFcInternalInterface`, `getFcSubComponents`, `addFcSubCompon` and `removeFcSubComponent`.

- *LifeCycleController* (to provide support for dynamic reconfigurations). This interface defines the following operations: `getFcState`, `startFc`, and `stopFc`. It corresponds to a minimal life cycle automaton (STARTED, STOPPED states).

It is possible to extend or to implement new interfaces, other than the ones provided. For example, some components may require very different life cycles. Of course, completely arbitrary life cycles can be specified by providing completely new interfaces, distinct from the LifeCycleController interface. More commonly, life cycles can be adapted from the basic one by extending the LifeCycleController interface to introduce new states and transitions or even to change the transitions of the basic life cycle. In this case, it is a requirement of the specification that the semantics associated to the STARTED and STOPPED states should be preserved.

2.1.4 Components Instantiation

Components in Fractal can be created in two ways. First, there are *factories*. The Fractal model distinguishes between generic component factories, which can create several kinds of component, and standard component factories, which can create only one kind of components, all with the same component type. Generic and standard component factories can provide the `GenericFactory` interface and the `Factory` interfaces, respectively.

Second, the components can be created using *templates*, which are a special kind of standard factory component that can create components that are quasi "isomorphic" to themselves.

As noted, components are created from component factories. To stop recursion, where component factories are created from component factories, a bootstrap component factory, which does not need to be created explicitly, and which is accessible from a "well-known" name, is necessary. This bootstrap component factory must be able to create several kinds of components, including component factories.

2.1.5 Typing

In Fractal a simple type system for components and component interfaces is used. This type system reflects the main characteristics of component interfaces, i.e. their name, their language type, and their role (client or server). It also introduces two new characteristics named contingency (indicates whether or not the functionality corresponding to this interface is guaranteed to be available while the component is running) and cardinality (indicates how many interfaces of type T a given component may have).

2.1.6 Summary: Conformance Levels Of The Specification

In addition, a Fractal component may provide or use new or alternative control interfaces, type systems, or even component semantics. For example, a Fractal component may provide a new *ConcurrencyController* interface to control concurrent accesses to the component. It may also provide an alternative *BindingController* interface, named for example *InternalBindingController*, to control the bindings between sub components directly from the enclosing component. It can also use an empty type system, with a unique type, sub type of itself, used for all components and component interfaces. A Fractal component may even define a new semantics for the communication between its sub components. For example, instead of specifying that operation invocations follow bindings, it can specify that operation invocations are broadcasted to all the sub components, in order to model an asynchronous communication.

The advantage of this extreme modularity and extensibility is that the Fractal component model can be applied to many situations. The drawback is that two arbitrary Fractal implementations might not be able to work together, because they will generally use very different, and potentially incompatible, options or extensions of the Fractal specification. In order to reduce this problem, sets of options called "conformance levels" are defined. Thus, it is possible to say that a given Fractal implementation conforms to the Fractal model of level X. Consequently, it is easy to check which Fractal applications and tools can work together, by comparing their conformance level to the Fractal model. The details are summarized in the Fractal specification [2].

2.2 Julia

Julia is the reference implementation of the Fractal component model in Java [3].

The main design goals of Julia include: the implementation of a framework to program component controllers, so that the user can freely choose and assemble the controller objects; to provide a complete continuum from static configuration to dynamic reconfiguration with flexible set of control objects, so that the user can make his own efficiency tradeoffs; and the implementation of these control objects to minimise the time overhead that these objects add to user applications and for the optimisation of the performance of the methods of the Fractal API.

2.2.1 General Data Model

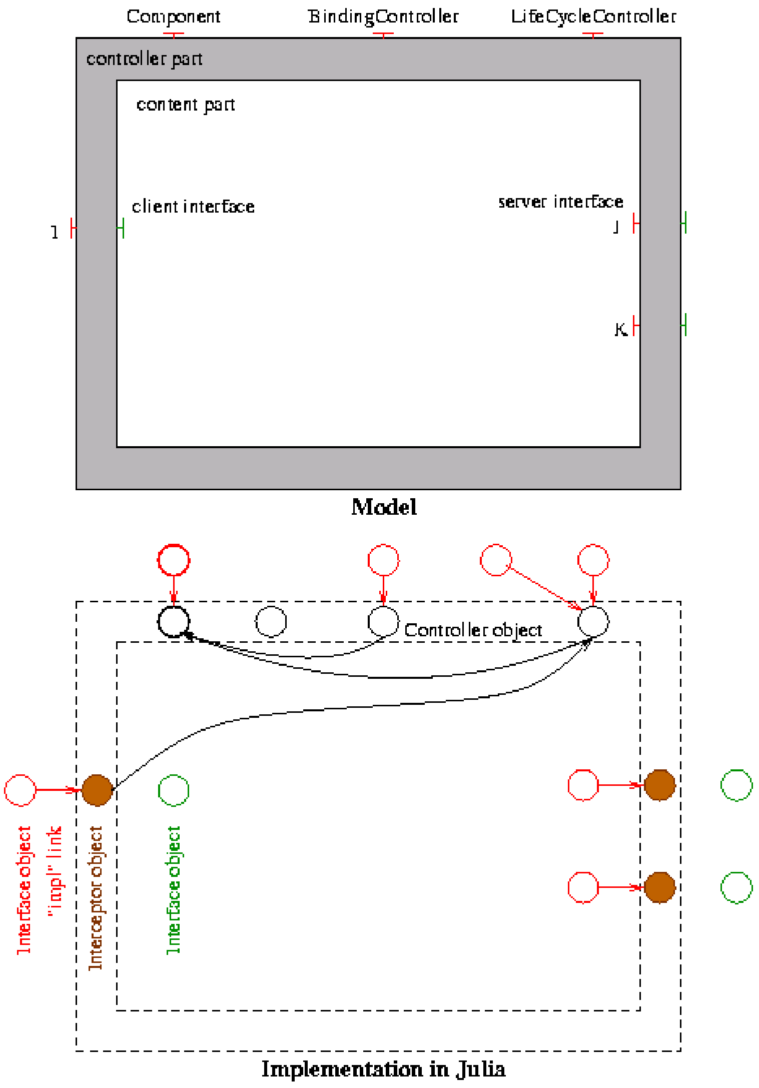


Figure 3: An abstract component and a possible implementation in Julia.

A general view of a Fractal component is presented in Figure 3. As mentioned above, a Fractal component is typically represented by many Java objects that can be separated into three groups, which implement: the component interfaces; the controller part of the component; and the content part of the component (not shown in the Figure).

The objects that represent the controller part of a component can be separated into two groups: the objects that implement the control interfaces (in black in Figure 3), and (optional) interceptor objects that intercept incoming and/or outgoing method calls on either the functional, business or user interfaces (in brown). These objects implement the *Controller* and the *Interceptor* interfaces, respectively. Each controller object can contain references to other controller objects (in black). Since the control aspects are generally not independent and must communicate between each other, they have references to each other.

In order to reconfigure a component, it is sometimes necessary to replace an object by another (for example to change an interceptor). However, in order to do so, the Java references to the replaced object must be changed and, in general, these references are not known. In order to be able to reconfigure Fractal components, Julia makes the following hypotheses: the objects encapsulated in container components (also called user objects) only see component interface objects. Furthermore, component interface objects cannot be replaced (but their internal state can be changed).

In order to be able to add protection mechanisms or to be able to provide distributed components on top of Julia (as Fractal RMI does), its authors decided to make some interfaces "hidden" (these start with "/"). This means that one cannot get the reference to such an interface without knowing its name. Further, although not yet implemented, it is also possible to perform access control checks in the `getFcInterface` method, to ensure that only Julia's classes can access these hidden interfaces.

2.2.2 Controllers in Julia

One of the main goals of Julia is to provide a framework to provide a flexible implementation of component controllers. To solve the modularity and extensibility problem in an effective way, Julia uses so-called *mixin* classes, which adopt an idea similar to multi-inheritance. A mixin class is a class whose super class is specified in an abstract way, by specifying the minimum set of fields and methods it should implement. A mixin class can therefore be applied (i.e. by overriding and adding methods) to any super class that defines at least these fields and methods. Moreover, the mixin classes used in Julia can be applied at runtime (unlike in most mixin based inheritance languages, where mixed classes are declared at compile time), and are therefore generated dynamically by the `MixinClassGenerator` class.

Mixin classes can be mixed, resulting in normal classes. More precisely, the result of mixing several mixin classes $M_1...M_n$, in this order, is a normal class that is equivalent to a class M_n extending the M_{n-1} class, itself extending the M_{n-2} class, ... itself extending the M_1 class. Several mixin classes can be mixed only if each method and field required by a mixin class M_i is provided by a mixin class M_j, with $j < i$ (each required method and field may be provided by a different mixin).

2.2.3 Interceptors

Julia makes it possible to generate interceptor classes at runtime, through byte code rewriting (by using the ASM package). The generator takes as parameters the name of a super class, the name(s) of one or more application specific interface(s), and one or more "aspect" code generator(s). It generates a sub class of the given super class that implements all the given application specific interfaces and that, for each application specific method, implements all the "aspects" corresponding to the given "aspect" code generator.

Each "aspect" code generator can modify the code of each application specific method arbitrarily. The order in which the aspects are"woven" is generally important, and must be specified by the user of the code generator.

Like the controller objects, the aspects managed by the interceptor objects of a given component can all be specified by the user when the component is created. Julia only provides two aspect code generators: one to manage the lifecycle of components, the other to trace incoming and/or outgoing method calls. Julia also provides two abstract code generators, named `SimpleCodeGenerator` and `MetaCodeGenerator`, which can be easily specialized.

2.2.4 Julia in distributed environment

The Fractal RMI components are a set of protocol, binder, and stub factory components that can be used to provide remote method calls between Fractal components. They can be used with any Fractal implementation.

In order to provide distributed bindings between Fractal components, Java RMI cannot be used, unless the Fractal Specification is changed to specify that each interface provided or required by a component must extend the java.rmi.Remote interface, and that all the methods of these interfaces must declare a java.rmi.RemoteException exception. Current implementations were unsuitable due to this and efficiency reasons.

Therefore a new, small and efficient, implementation of JavaRMI in Fractal has been deployed. Moreover it does not need a static stub compiler, and does not impose constraints on remote interfaces, unlike Java RMI (the Interface interface plays the role of the Remote interface: the objects that implement this interface, namely the interfaces provided by Fractal components, are passed by reference, while all other objects are passed by value).

Writing an application that uses Fractal RMI is extremely simple, as it practically does not differ from writing a normal Fractal application. The only difference is that some components are denoted as “virtual-nodes”, which means that they are remote. Additionally, a name to this virtual node is provided. The binding between virtual nodes and hosts is then performed by clients, who look up available hosts and associate virtual nodes with them.

In order to establish the connection, the server (host) registers itself under some name in the naming service:

```
NamingService ns = Registry.getRegistry("my-server");
ns.bind("server-host", Fractal.getBootstrapComponent());
```

The registered component of the server is then looked up and used by the client:

```
Map context = new HashMap();
context.put("remote-node", ns.lookup("server-host"));
```

2.2.5 Component structure modification

In Fractal it is very easy to modify the structure of the components. For example, to change the sub-component A to A’ of a component B one should do the following:

1. instantiate the new sub-component A’
2. find the component B in the whole application
3. find any components that are bound to component A, e.g. C
4. get the content controller of component B
5. add component A’ as sub-component of B
6. get the life cycle controller of both component B and C if their bindings to A were mandatory
7. get binding controller of component B
8. stop components B and C
9. unbind A
10. bind A’
11. start components A’, B and C

FractalExplorer The Fractal Explorer console is a tool that provides a way for reconfiguring and managing Fractal-based applications at runtime [4]. It is a graphical tool, which allows one to access all features offered by the Fractal framework and components (e.g. discovering the components of the application and introspecting their structure; managing the controllers; monitoring the application status; and reconfiguring the application at runtime by updating bindings etc.).

The Fractal Explorer is in fact a Fractal component. It requires binding the Component interface of the Fractal-based application.

2.3 ProActive and the hierarchical Grid components

ProActive [6] is a Grid middleware for parallel, distributed, and concurrent computing. It features mobility and security in a uniform framework. ProActive is implemented in Java and uses standard Java classes, which do not require changes to the Java Virtual Machine. Overall, it simplifies the programming of applications distributed over Local Area Network (LAN), Clusters, Intranet or Internet Grids.

The main architecture concept of ProActive is the *active object*. It is the basic unit of activity and is the basis of the distribution mechanism for building concurrent applications. An active object owns its own thread. This thread only executes the methods invoked on this active object by other active objects and those of the passive objects of the subsystem, which belong to this active object. With ProActive, the programmer does not have to explicitly manipulate Thread objects, unlike in standard Java.

2.3.1 ProActive as a component framework

The most interesting part of ProActive in the context of our report is the parallel and distributed component framework for building Grid applications [7].

Although, this framework is an implementation of the Fractal specification, it was not possible to use Julia, the reference Fractal implementation. The reason for that was that Julia manipulates a base class by modifying the bytecode or adding interception objects to it. On the other hand, ProActive is based on a meta-object protocol and provides a reference to an active object through a typed stub. If active objects with Julia were to be used, the Julia runtime would try to manipulate the stub, and not the active object itself. And if trying to force Julia to work on the same base object than ProActive, the control flow could not traverse both ProActive and Julia.

Consequently, ProActive created own implementation of the Fractal specification. This implementation is conformant up to level 3.2 accordingly to Fractal classification. In other words, it is fully compliant with the API, except it does not consider the creation of components through template components. The ProActive implementation is different from Julia both in its objectives (Grid and P2P environments) and in the programming techniques.

2.3.2 Extensions to Fractal

ProActive introduces two extensions to the base Fractal model: distributed deployment and parallel components.

The component model in ProActive leverages the automatic deployment capabilities of the system. Distribution is achieved in a transparent manner over the Java RMI protocol due to the use of a stub/proxy pattern. A component is manipulated independently of its location (local or on a remote JVM). Additionally, the deployment framework allows for automatic creation of distributed components. Using a configuration file and the concept of virtual nodes, the framework connects to remote hosts (using rsh, rlogin, ssh, globus, . . .), creates JVMs, and instantiates components on them.

A parallel component (aka Grid component) is a specialization of the Fractal composite component – see figure 4.

Grid components encapsulate other components of the same type, and all incoming calls are forwarded to the corresponding internal interfaces of the enclosed components. This allows parallel processing while just manipulating one entity, the enclosing parallel component. Additionally, by using the typed groups API of ProActive, coupled with the concept of internal collective interfaces of Fractal, the communications to the enclosed components are either scattered or broadcasted.

3 Building mediator components with the Fractal model

Our goal was to investigate the Fractal model in order to implement the Mediator Toolkit using components. To gather experience on working with Fractal we decided to experiment with a typical use case, a Successive over Relaxation problem (SOR). We will describe the SOR use case in the following section. Here, we first outline our findings about building a mediator component toolkit.

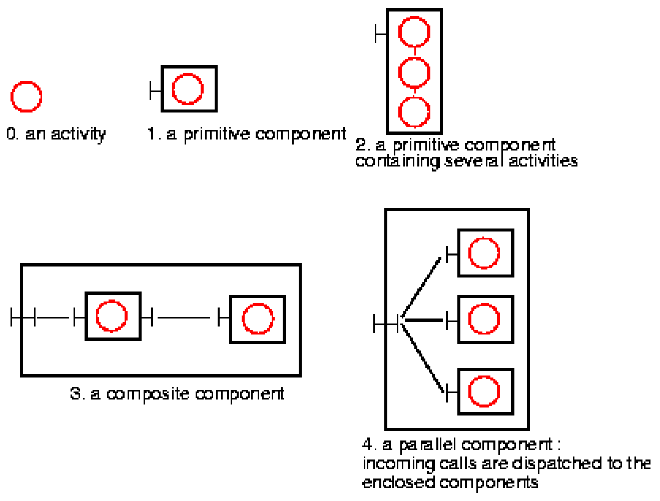

0. an activity
1. a primitive component
2. a primitive component containing several activities
3. a composite component
4. a parallel component : incoming calls are dispatched to the enclosed components

Figure 4: Different kinds of components with the ProActive implementation

3.1 Mediator Toolkit

The existing model of Mediator Toolkit is presented in Figure 5.

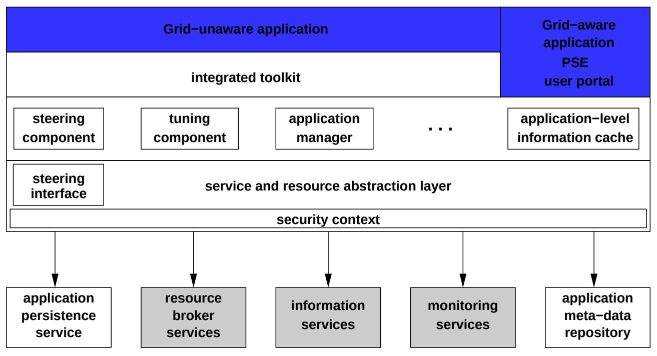


Figure 5: Existing model of the Mediator Toolkit.

Based on experience gathered when investigating the Fractal component framework, we propose the following extensions. First, in order to effectively modify the structure of a running application, we propose to implement an "explorer component". Thus, the user could switch between different implementations of his/her algorithms without the need to stop and re-run their application.

Second, we believe that the Mediator Toolkit can greatly benefit from implementing different control aspects of the application separately, namely by using controllers. We propose to introduce into the architecture the following controllers, as depicted in Figure 6:

- steering – for modifying application parameters, which would allow for computational steering during runtime
- persistence – for handling snapshots: initiating snapshots, as well as starting (from snapshot or from scratch) and stopping the application
- distribution – for optimal utilization of allocated resources, and for adapting to changes in environment (loosing and acquiring resources, changes in quality of network connections)

- component – for investigating the application's structure (in terms of components) and modifying it (e.g. switching to alternative implementation, replacing subcomponents)

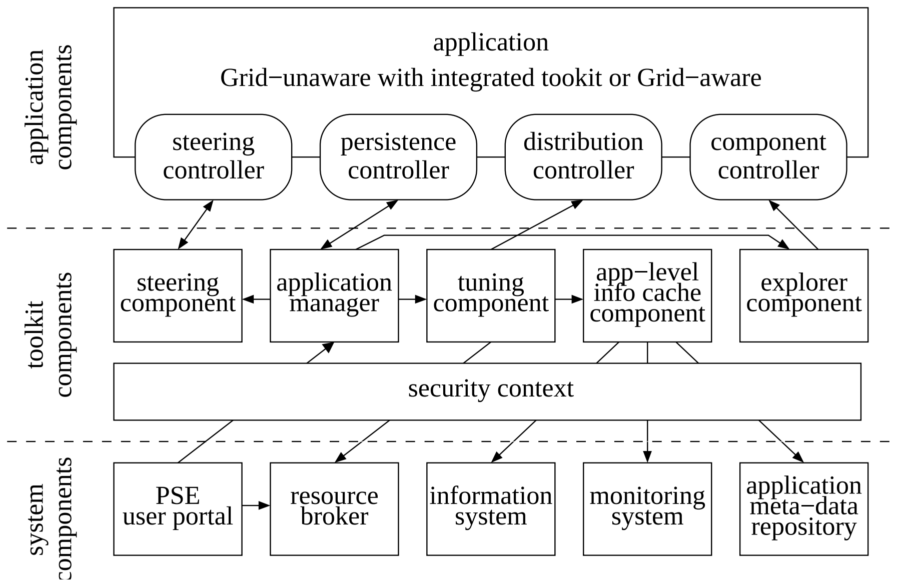


Figure 6: Generic component platform

Note that the component controller is already implemented in Fractal. However, the other controllers have to be designed according to the necessary functionality.

Another important observation is that communication with the application is only via its controllers. This can be exploited when designing constraints on the application, as we discuss below.

3.2 Overview of the prototype implementation

As mentioned above, we used a SOR example in order to gather experience about using Fractal framework to implement Mediator Toolkit. Our **SOR-application** consists of a master and a set of workers. The master is responsible for data distribution and management. It decides whether to continue calculation or stop it and commands on making snapshots. Workers, in turn, contact the master to acquire data for processing and for returning results to.

The communication between master and workers, as well as between workers themselves is implemented using Ibis RMI.

The master uses JavaGAT to submit workers to the Grid (we used VU's DAS-2 cluster during the experiments).

Finally, the entire application (master and workers) is encapsulated into a JAR file, which is provided for execution to the Mediator Toolkit. The **Mediator Toolkit** is a Fractal application.

3.3 Persistence Controller and Life Cycle Controller

The *Persistence Controller* is the manager of application's instance. Not only is it responsible for snapshoting, but also for starting (from scratch or from snapshot) and stoping application. For this to be accomplished, we propose bounding the Persistence Controller with the Fractal's *LifeCycleController*. The latter is a simple state automaton (with two states: `started` and `stopped`). We propose extending the state-cycle to service snapshoting. See figure 7.

We propose to extend the `started` state of the component by adding substates representing different stages of the running application (`created`, `initialized`, `running`, and `finished`) and its snapshoting (`snapshoting`).

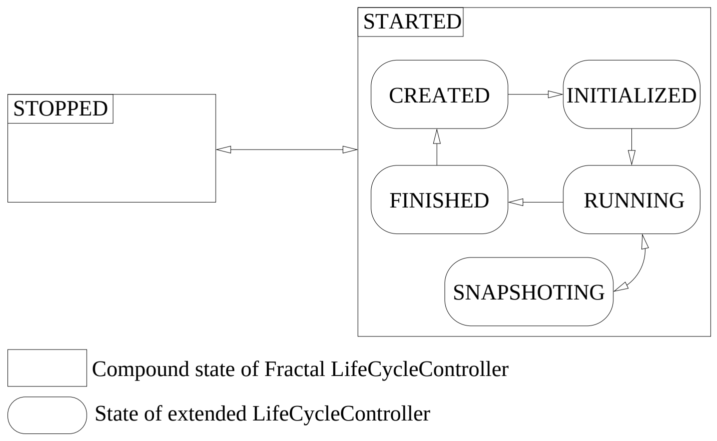

STARTED
STOPPED
CREATED
INITIALIZED
FINISHED
RUNNING
SNAPSHOTING
Compound state of Fractal LifeCycleController
State of extended LifeCycleController

Figure 7: Extended states of LifeCycleController

The Fractal LifeCycleController is responsible for starting and stopping the component. There are certain conditions under which a component can be stopped. For example, all method invocations on this component should have finished (special interceptor keeps a counter of active method invocations). Similarly, only a component with all mandatory interfaces bound can be started.

Our system could also benefit from this approach. Transitions between `stopped` and `started` states could be limited to only a few `started` substates. The component should not be allowed to stop when the snapshoting is being performed. Additionally, stopping an application in the `running` state could mean interrupting the application (transition to `finished`) first.

3.4 Application controllers

The proposed application controllers (steering – *sc*, persistence – *pc*, distribution – *dc*, and component – *cc*) may be implemented either as Fractal-controllers (part of the membrane) or as usual components – see figure 8.

3.4.1 Application controllers as part of the membrane

Application controllers can be implemented in the same way "native" Fractal controllers are implemented in Julia – by using Mixin. New controllers can extend utilize multiinheritance to enable close cooperation. Controllers implemented in such a way would be placed inside the components membrane – see figure 8(a).

Additionaly, this approach could benefit from components types in Julia (e.g. primitive, composite). We could define a new type of component to provide the applications developers with a simple way of encapsulating their application into a framework-enabled component.

Because this approach isolates the configuration (which involves steering, distribution, persistence and component controllers) and implementation, it seems to fit perfectly into the overall Fractal architecture. However, it requires rather complicated implementation (using Mixin) for both – the mediator toolkit developer and the application programmer. The mediator toolkit developer has to use Mixin in order to implement default controllers. The application programmer, in turn, is forced to use Mixin to prepare custom controllers.

Investigating this approach is left for future work.

3.4.2 Application controllers as components

In this approach, controllers are implemented as usual components – see figure 8(b). They export server interfaces to be connected to the framework and client interfaces for binding with the application.

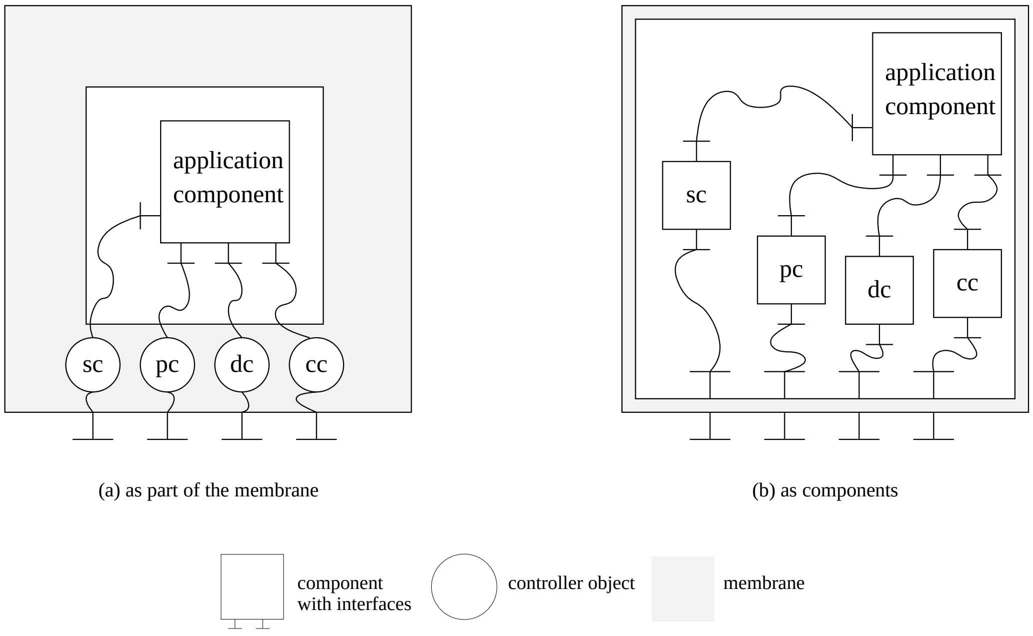


Figure 8: Controllers

Using typical components significantly eases the implementation. At the same time, it does not violate the Fractal architecture. Controllers can be simply regarded as the intermediate layer of components between the application and the framework.

This approach will be discussed in broader context in the next section.

3.5 Dealing with different types of application

The Fractal Mediator Toolkit is ready to run not only with applications following the Fractal design, but also with non-Fractal ones – see figure 9.

3.5.1 Non-fractal-like applications

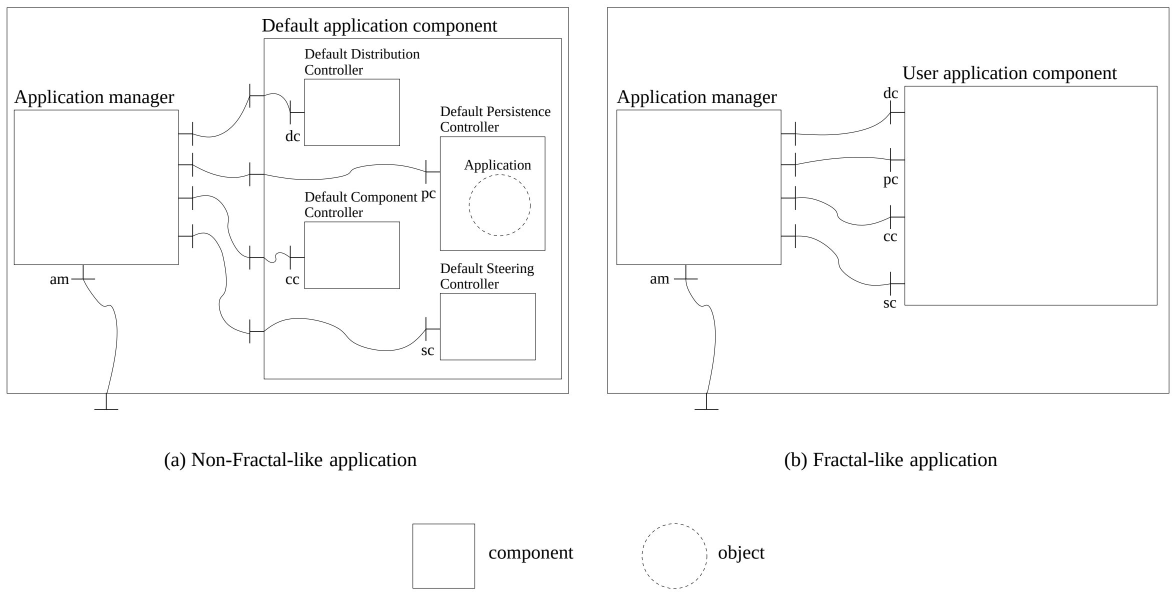
The framework is able to cooparate with applications that do not use the Fractal framework. In that case, the set of default controllers is created.

A user application is encapsulated into the default persistence controller as this component is responsible for starting and stopping the application, and it has direct access to the application object. This controller, together with default distribution, component, and steering controllers, are integrated into a default application component, which is bound to the rest of the framework via the application manager.

The default implementations of controllers are very simple. Only the persistence controller is able to perform some actions – starting and terminating the application. All other methods in this and the remaining controllers throw the ‘not yet implemented’ exception, as they cannot be provided without a specific knowledge about the application.

3.5.2 Fractal-like applications

The Fractal-aware applications are very easy to connect to the framework.


Default application component
Default Distribution Controller
Application manager
dc
Default Persistence Controller
Application
Default Component Controller
pc
cc
Default Steering Controller
am
sc
User application component
Application manager
dc
pc
cc
am
sc
component
object

(a) Non-Fractal-like application

(b) Fractal-like application

Figure 9: System design

The only requirement towards the application developer is to deliver a Fractal component ('user application component', see figure 8(b)) with exported interfaces for each of the controllers (`dc`, `pc`, `cc`, and `sc`). Internally, they are expected to be bound to the user's implementation of the controllers.

3.6 Security

An interesting possibility for the framework could be using the interceptors to implement the security.

Interceptors provide the ability to surround any method invocation with virtually any code, including exception handling. Interceptors are part of the membrane. They can communicate with controllers.

A possible architecture could involve inserting security interceptors for all calls outgoing from the application. Additionally, the application component membrane could be enriched with a security controller. In order to use Grid services (make external invocations), the application would have to provide the security controller with proper credentials, which would be used by the interceptors on each method invocation to verify access.

4 Conclusions

This report summarizes our experiences with investigating the Fractal component model. We have described Fractal specification and its three implementations. Julia is the reference implementation, which we later used during our experiments. However, we also mentioned interesting alternative implementations: aspect-oriented AOKell and Grid-oriented ProActive.

Our main goal has been to gather experience on using Fractal as an example component framework, when designing a Mediator Toolkit in a component manner. The important improvement over the current Mediator Toolkit design have been controllers. More precisely, we have introduced four controllers (component, parameters, persistence, and distribution) in order to separate different aspects of the application control. Furthermore, we have presented how such controllers could be implemented in Fractal. We have also shown that the Mediator Toolkit could provide support for both component and non-component applications so that all types of applications could be run on the Grid.

References

[1] http://fractal.objectweb.org/specification/index.html

[2] http://fractal.objectweb.org/specification/index.html#tth_sEc7.1

[3] http://fractal.objectweb.org/current/doc/javadoc/julia/overview-summary.html

[4] http://fractal.objectweb.org/tutorials/explorer/index.html

[5] http://fractal.objectweb.org/tutorials/aokell/index.html

[6] http://www-sop.inria.fr/oasis/proactive/

[7] F. Baude, D. Caromel, M. Morel, *From Distributed Objects to Hierarchical Grid Components*, International Symposium on Distributed Objects and Applications (DOA), November 2003
http://www-sop.inria.fr/oasis/ProActive/doc/HierarchicalGridComponents.pdf

An ADL-based Support for CCA Components on the Grid

Maciej Malawski, Tomasz Bartyński, Eryk Ciepiela,
Joanna Kocot, Przemysław Pelczar, Marian Bubak
Institute of Computer Science and ACC CYFRONET
AGH University of Science and Technology
Kraków, Poland

Abstract

Moccaccino project described in this paper provides a novel ADL specification and a manager tool that facilitates the deployment and management of component applications on the Grid. The concept of hierarchical component groups and connections indexed as lists or maps is introduced in the ADL. The designed and implemented Moccaccino Manager is suited for MOCCA, a CCA compliant framework build upon H2O, which is a secure and lightweight middleware platform providing a flexible component container. The interoperability with GCM and CoreGRID research group on component deployment is addressed as well.

1 Introduction

Component-based programming paradigm becomes increasingly popular model in scientific computing [6]. A component model supports software composition, deployment and reconfiguration, which are the essential requirements for building and running applications on such environment as Grid. Running component based applications on the Grid environment still remains a challenging problem involving resource discovery, component deployment and application management in response to changes of environment.

The main characteristics of the problem may be briefly summarized as follows:

- The application components and their connections may have various demands for computing power and communication cost;
- The application may include collections of components;

- The number of components in a collection may depend on the number of available computing nodes;
- The number of nodes may change in time.

The main assumption is that the component application should be possibly unaware of the complexity of the environment it is running on, so the responsibility for dealing with deployment and adaptation process should be thus delegated to the specialized manager tool.

Consequently, such a manager tool should be aware of both the environment and the application structure. Environment awareness can be achieved by using an appropriate discovery and monitoring techniques. The component application structure respectively can be expressed using an Application Description Language (ADL). In this paper, we describe our approach to development Moccaccino: a manager for a component application running on the Grid and the ADL designed for this purpose. The background for our research is the CCA [2] model and its distributed framework MOCCA [9, 8].

The advantage of the CCA modes is that it was designed to satisfy the requirements of scientific applications. Moreover, the simple specification and support for reconfiguration of application at runtime make CCA useful basis for our research. MOCCA is a CCA compliant distributed framework, based on H2O platform which offers lightweight component containers called H2O kernels. Current version, called MOCCA_Light, is a pure-Java implementation and provides some extensions of CCA for multiple ports and connections.

2 Related work

A number of projects is being developed, that aim in supporting automated deployment of component applications in Grid environment. In order to fulfill this objective, they produce their own Architecture Description Languages.

The Fractal ADL [3] defines the component types, their connections and containment relationships between hierarchical components. Recent work on extending the Fractal towards the Grid Component Model also addresses the deployment problem and proposes a common solution for automating this process [4].

Grid Application Factory Service [5] for CCA addresses the problem of building reliable, scalable Grid applications. This solution is based on separating the process of deployment and hosting from application execution, and uses XML documents for describing these stages.

Corba Component Model (CCM) also has the concept of IDL3 language, which is responsible for application assembly and deployment. Within the Grid-

CCM project there exist a solution for deploying Corba components on the Grid infrastructure [7] using Globus.

In addition to the examples presented above, other ADLs were inspected and taken into consideration. A good overview of ADLs can be found in [1].

Despite the number of ADLs, none of them fully satisfies our requirements. Problems such as describing components characteristics (computational and communicational needs) or building structured component-based applications are not solved. Furthermore, specifying number of components in a way that would enable adapting quantity of components to available resources is not supported.

3 Our approach - ADL concept

Considering the advantages, disadvantages and limitations of above-mentioned ADLs we identified the following goals to achieve when designing the ADL for Moccaccino (ADLM):

1. to make ADLM as concise as possible,
2. to facilitate modification of the quantitative properties of architecture,
3. not to lose expressiveness of the language, therefore to support a breadth of application architectures,
4. to link resources such as executables and documents using URLs,
5. to allow containing information related to other aspects of deployment such as scheduling or policies of adaptation.

To facilitate document processing and programming the tools the XML-based syntax was developed. The main elements of the description are described below. *Component class* is a basic element that defines type of a component. To facilitate the parametrization of application description the term of *component instances group* is introduced.

Component instances group is established by zero or more component instances. These instances are physically separate and independent, but their connections follow the same pattern. For example in case of master-worker architecture, every single worker is connected with master, therefore despite that workers are individual they are connected according to the same pattern. If so, we may logically treat them as a *component instances group* and represent it as an ensemble in the *qualitative component diagram*.

When a component is connected to a collection, there is a need for an addressing scheme for indexing the multiple connections. *Connection qualifier* defines the addressing scheme in pluggable mechanism, and such basic qualifiers as list or map are provided. The connections are parametrized by qualifier-specific properties such as list length or map key set.

The important rule in our system is that the connection multiplicity implies how many of component instances of a group are to be created. For example, if a master component is connected to the workers collection using a list qualifier, than the number of workers will be equal to the list length.

Obviously, since the *qualitative component diagram* may not be a simple tree, the connection cycles may exist. It may result in the unambiguity of the number of component instances. This issue amongst the others is covered by Component Graph Builder (CGB) which validates the diagram.

Moccaccino can support as well ordinary quantitative component diagrams by describing all connections as individual (with multiplicity equal to one) and then treating *component instances group* as a single component instance.

It is also worth noting that in ADLM number of Moccaccino-specific decorations are present. These include attributes associated with *component instances group* and connections which provide information about computational power needs and connection load which are used for deployment optimization.

4 Architecture of Moccaccino

In order to validate the proposed ADL approach, the Moccaccino Manager which was designed and implemented. Figure 1 presents the Moccaccino components' dependencies along with results of their activities.

The Moccaccino Manager is responsible for deployment, execution and management of the application. Its functional components are: ADLM Unmarshaller, Component Graph Builder (CGB), Kernel Information Provider (KIP), Deployment Planner (DP) and Application Deployer (AD).

The ADLM Unmarshaller component is responsible for parsing the file that contains the application description ADLM file (XML-based). While parsing, the application model is built using Component Graph Builder to create components graph. The model that represents application is called Application Object Model (AOM). It is a data structure that can be exchanged between functional components to construct and deploy the application.

The deployment process itself is handled by the Application Deployer component. The result of his action is a deployed application and a handle to it, which Manager can further use. Application Deployer uses a Deployment Plan, which assigns H2O kernel to each application's MOCCA component. This plan is built by the Deployment Planner, that can implement different policies to make the plan possibly optimal. The H2O kernel information needed for the construction for

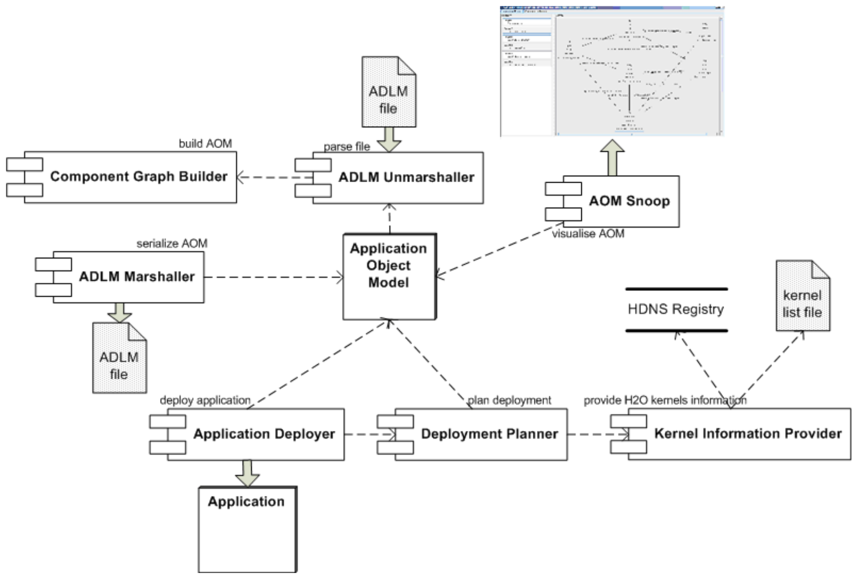


Figure 1: Moccaccino components

Deployment Plan are obtained from the Kernel Information Provider component which may be configured to run in static mode - by providing a list of available kernels or dynamic mode in which case it uses the HDNS registry.

Moccaccino provides also a tool for visualizing the application model, expressed either in AOM or ADLM form.

5 Integration within the STE Institute

The approach described here fits well in the context of the Systems, Tools and Environments Institute of CoreGRID mainly for the reason that it follows the generic component paradigm. There are also two more specific integration activities, which strengthen the relevance of the presented work within the Institute.

The first is the response to the newly drafted GCM (Grid Component Model) specification. Our recent work shows that the CCA and GCM components can be integrated into one component system, and moreover two frameworks: one CCA-based as MOCCA and another Fractal-based as ProActive can interoperate at runtime and build one large distributed component applications.

The second one is the activity of the Application Deployment Component research group. The Application Manager tool and ADL-based description within Moccaccino project as an alternative solution bring a new experience to this research group and helps better understand and analyze the process of component deployment on Grid.

6 Summary and future work

In this paper we presented Moccaccino: an application manager tool for supporting CCA components on the Grid. The tool required developing a new Application Description Language, which supports hierarchical parametrized component groups and connections as well as performance hints for optimizing the deployment. The current and future work includes development of deployment and adaptation algorithms, introducing on-line monitoring support, tests on real applications and further integration with deployment solutions within GCM.

Acknowledgments: This work was supported by EU IST CoreGRID project and Polish grant SPUB-M. Maciej Malawski acknowledges the support from the Foundation for Polish Science.

References

[1] Architecture description languages. http://www.sei.cmu.edu/architecture/adl.html.

[2] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren. The cca component model for high-performance scientific computing. *Concurr. Comput. : Pract. Exper.*, 18(2):215–229, 2006.

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

[4] M. Coppola, M. Danelutto, S. Lacour, C. Perez, T. Priol, N. Tonellotto, and C. Zoccolo. Towards a common deployment model for grid systems. In S. Gorlatch and M. Danelutto, editors, *CoreGRID Workshop on Integrated research in Grid Computing*, pages 31–40, Pisa, Italy, November 2005. CoreGRID, IST.

[5] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. G. L. Ramakrishnan, and A. Slominski. Grid web services and application factories.

[6] V. Getov and T. Kielmann, editors. *Component Models and Systems for Grid Applications.* Springer, 2005.

[7] S. Lacour et al. Deploying CORBA components on a computational grid. In W. Emmerich et al., editors, *Component Deployment: 2nd Int. Working Conf., CD 2004, Edinburgh, UK, Proc.*, volume 3083 of *LNCS*, pages 35 – 49. Springer, 2004.

[8] M. Malawski, M. Bubak, M. Placek, D. Kurzyniec, and V. Sunderam. Experiments with distributed component computing across grid boundaries. In *Proceedings of the HPC-GECO/CompFrame workshop in conjunction with HPDC 2006*, Paris, France, 2006.

[9] M. Malawski, D. Kurzyniec, and V. Sunderam. MOCCA – towards a distributed CCA framework for metacomputing. In *Proceedings of the HIPS2005 Workshop in conjuncion with IPDPS 2005, Denver, USA, April 2005*. IEEE, 2005.

Communication Models for Processes and Services in Mobile Lightweight Grid Systems

L. Kirchev[1], S. Isaiadis[2], V. Georgiev[1],V. Getov[2]

[1] Institute for Paralel Processing, Bulgarian Academy of Sciences
{vasko, lkirchev}@acad.bg
[2] Harrow School of Computer Science, University of Westminster
{s.isaiadis, v.s.getov}@westminster.ac.uk

Abstract

The last few years we have seen the emergence of pervasive and mobile computing technologies. At the same time the Grid has become the de facto standard for distributed and high performance computing. A lightweight Grid infrastructure may well provide a solid foundation for pervasive computing. For this to happen, we must rethink our design goals, and provide support for mobility, resource limited and non-dedicated environments, and redesign the process communication models to reflect the dynamic conditions of the system. In this paper we try to identify the requirements for supporting service and session mobility in lightweight Grid systems; review current approaches and discuss their limitations; and present our ideas for an integrated platform that meets these requirements.

1. Introduction

Applications deployed on computational grids very often require communication facilities, as is the case, for example, with synchronous or interactive parallel applications. This necessitates the grid framework to offer relevant supporting communication mechanisms. An issue that is related to communication is the application task migration. In SOA environments, an application may be composed by a number of services in a workflow, so actually a task migration may be service migration as well.
Task migration may be triggered by a resource or service failure in the Grid infrastructure, or imposed by the load balancing mechanisms in the system.
The need for migration schemes becomes even more obvious in such dynamic environments as Grid systems comprised mainly of mobile devices. The latter are much more volatile than traditional Grid resources, as they are susceptible to a wider ranger of failures such as increased energy sensitivity due to battery issues; intermittent connectivity and unpredicted connection patterns due to the unreliable wireless links; and possible resource constraints due to size and mobility restrictions.
In this paper we take a fresh look at mobile and lightweight Grid infrastructures. Our perspective puts the mobile devices in the center and we discuss a set of requirements necessary to realize our conceptual architecture. We present a number of relevant projects in the field of checkpointing and migration and then propose our solution. Finally, we conclude this paper and lay the roadmap for our future work.

2. Relevant Work

ProActive [1] is a Java library for developing distributed and grid applications. It is based on the concept of the active object – i.e. object that has its own thread of execution. The application is built up of subsystems, any of which have a single active object and may have also passive objects. Outside the subsystem, only the active object is visible.

ProActive supports active objects' mobility, grouping of objects and establishing communications between the objects in the group, and also a SPMD programming model. Any active object in ProActive may migrate to another location (another JVM). The migration may be triggered either by the object itself, or by another agent. If the active object that migrates references passive objects, they will migrate with it too. After migration of an active object, messages sent to it are forwarded to the new location of the object.

ProActive offers a flexible mechanism for creating groups of objects and performing operations on the group as a whole. The results of such operations are also groups. The group communication enables simulation of an MPI-style collective communication. This can be used for implementing SPMD activities. Here an SPMD group is a collection of objects which reference all the other active objects in the group.

Ibis [2] project is a Java-based programming environment for grid computing. Central for its architecture is the Ibis Portability Layer (IPL), which offers an interface to different grid services such as monitoring, resource management, etc. It can have different implementations which may be selected and loaded at run time.

IPL provides the interface to various communication mechanisms. It provides one basic communication abstraction – unidirectional message channels. The endpoints of communication are send and receive ports, which are connected through these channels. For group communication, a send port may be connected to multiple receive ports, and vice versa. The actual communication may be realized through different protocols, such as TCP, UDP, MPI and specialized protocols such as Panda and GM, where supported by the interconnection. Ibis implements several application programming models on top of the IPL – RMI, whose implementation has some optimisations over the traditional RMI, group method invocation (GMI), which extends RMI with group communication, divide-and-conquer parallelism in the Satin system, and RepMI for communication through replicated objects (RepMI is under construction).

Although Ibis offers variable communication facilities, it does not have enough support for service mobility in highly dynamic environment. The Satin system offers migration, fault-tolerance and malleability, but for a specific type of problems – the divide-and-conquer tasks, while for a grid with clusters of mobile devices a more general model is needed.

Agents based Grid systems. Many grid systems use some form of mobile agents. As descried in [3], in a grid system composed of desktop computers a user might be represented by an agent, which the user receives upon registering in the grid system. The user submits jobs in the system through the agent, which finds machines for execution of the job. If more machines are needed, the agent spawns child agents and sends them to different computers, where they start and monitor the user process. If a machine becomes unavailable, the monitoring mobile agent migrates the process on another available machine. For this purpose checkpointing is used.

For the purpose of process interaction, each process involved in the same job is addressed with a sequential number, valid only inside this job. Upon migration, this ID is translated into appropriate IP address. A Java wrapper object is in charge of supporting the process interaction. The user process calls the wrapper's message-forwarding method, passing the ID of the destination process. The message is passed to the corresponding mobile agent that tracks the machines, participating in the job execution. It maintains a table which translates ID to IP address. Upon migration the mobile agent sends the new address only to the processes that have communicated with the migrated process within a certain time interval.

JGrid is a Java and Jini-based grid system, developed at the Veszprem University, Hungary. The system supports process interaction based upon MPI-like message passing and high-level remote method calls [4], [5].

For the first one Java MPI method calls are used for communication. The Compute Services create the processes that perform the computations and establish socket connections between them. For the mapping between the physical channels and the logical connections a translation table is used. The mapping is transparent to the working processes. After starting the program, MPI_init method initializes the communication infrastructure, thus connecting the processes through logical channels. In this way a wide-area parallel system may be crated.

The second mechanism for process interaction allows tasks to communicate through remote method calls made through task proxies. When a remote process is created, the client receives a task control proxy, which references the task and may be passed to other tasks. Thus a set of remote tasks may store references to each other. Tasks can call remote methods on other tasks to implement a desired communication model.

Service migration is supported by the JGrid system through application level checkpointing – the service that requires migration support implements an interface with methods for saving the current state of the service. In case of migration the system is responsible for restarting the service on another host from the last checkpoint.

H2O [6] is a component-based and service-oriented framework, intended to provide lightweight and distributed resource sharing. It is based upon the idea of representing the resources as software components, which offer services through remote interfaces. Resource providers supply a runtime environment in the form of component containers (kernels). These containers are executed by the owners of resources and service components (pluglets) may be deployed in them not only by the owners of the containers, but also by third parties, provided that they possess the proper authorization. Inter-component communications are based on an extension to the Java RMI, called RMIX. It supports asynchronous calls, one-way calls and dynamic choice of the wire protocol.

H2O allows for the creation of a communication infrastructure that facilitates the execution of MPI programs across multidomain clusters. "Proxy" pluglets are used for transparently intercepting and forwarding the messages to the remote site. Proxy pluglets benefit from the H2O communication facilities for forwarding messages. They are loaded on all participating kernels and handles to these pluglets are distributed among all kernels. The communication between proxy pluglets is performed through direct channels between pluglets. The implementation of this infrastructure uses the MPICH library and makes some enhancements to it.

H2O does not offer direct support for service migration.

3. Requirements for Service Mobility

The very different nature of mobile devices -and the subsequent integrated mobile Grid systems, mandates that we must rethink our design goals of the fundamental infrastructure layer of the Grid system. Supporting mechanisms in such environments are very different than traditional ones. We can identify mainly three classes of requirements in this case: monitoring related, group related and migration related requirements. In the following paragraphs we discuss in more detail these requirements.

3.1. Monitoring

As they say, prevention is better than cure, and so we must take actions to at least try to anticipate possible failures in our systems –if not prevent them altogether. Prevention is arguably a very challenging task, but in our case, diminishing the impact that a failure can have on our system is almost equivalent. For this to be realized, we need a monitoring scheme that will generate notifications in case a condition that might lead to a failure occurs in a resource or a particular service.

There are two classes of events that we need to monitor in such a system: hardware resource loads and connectivity issues. Hardware resources need to be monitored because they may lead to an overloaded resource and hence reduced performance. In an networked environment, reduced performance becomes the equivalent of a failed resource. Keeping tags on the loads of the most important resources –CPU, memory, network, battery, will allow us to take preemptive action.

Connectivity issues will eventually arise due to the mobile nature of the participating hosts. Trying to predict the mobility patterns of the resources/services might give us an opportunity to save and restore a task in a different node instead of helplessly waiting for the disconnection.

3.2. Checkpointing / Migration

Checkpointing and migration mechanisms become a necessity in very dynamic and unreliable environments. A mobile Grid system should support both transparent migration of services and application tasks in other available hosts. Neither the clients nor the service itself should have knowledge of the migration procedures.

In order for migration to be usable, checkpointing functionality should be supported as well. A snapshot of the executing task should be taken at periodic times, detailed enough for the migrated service to be able to continue from that point instead of starting all over again.

3.3. Group Formation and Communication

Formation of groups is needed for collective communications inside the mobile Grid system. Such communication is needed to identify and query available nodes; to collect information regarding the status of the nodes; to identify and select possible migration targets; and generally to realize and support the mobile service communication model. The formation of the groups should be truly dynamic –participating services should be able to join and leave the group at any time and for any reason.

4. A Lightweight Grid Communication Model

The key approach we see as crucial for meeting the aforementioned requirements is the grouping and clusterification of the non-dedicated and often inadequate resources in order to split the system functionalities in related layers and abstract away from prospective clients the mobile and dynamic nature of these resources.

In a previous work [7] we developed a communication model for a hierarchical lightweight grid platform. Its architecture consists of three layers. The different clusters which comprise the system represent the first level. The second level is the grid level which is comprised of inter-connected clusters. The (optional) third level is the ability to connect the platform with other gird systems. The entry point for both cluster and grid level is a portal. We have cluster portal for each cluster and a grid portal for the interconnected clusters. There are system services local for each cluster and they have corresponding representatives on the grid level.

The cluster level services are represented in the outer Grid through a number of aggregator services deployed at the cluster level proxy. One aggregator service is automatically generated and deployed for each group of similar service in the cluster (where similarity is determined by the published interface) These present a uniform and consistent interface to a dynamic set of underlying services. It also provides location and binding transparency which in turn gives us the flexibility to allow for many different binding protocols to be used for the cluster services e.g. Web or Grid services, RMI Activatable Objects and so on. The proxy knows all the details about the underlying nodes and hence clients wishing to communicate with services available in the cluster have to pass through the proxy. They will have no a priori knowledge of the topology, communication protocol, or binding address for the services –the proxy handles this procedure.

The communication among running services/tasks requires that they be uniquely identified. One obvious solution is to introduce unique ProcessID (TaskID) within the cluster and a ClusterID for each cluster in the grid. Thus ClusterID+ProcessID points to a single running task. When Resource Management Service (RMS) receives a task for execution and allocates it to a particular node, it creates the mapping ProcessID-NodeID-ClusterID.

4.1. Communication Management

The platform uses centralized management (cluster-wide but not grid-wide) based on a system Communication Service (CS). CS performs similar activities to those performed by a communication meta-service with master-workers model. In this scenario messages are sent to CS for particular process and CS forwards them to the exact location. CS interacts with the Monitoring Service (MS) and RMS in order to update information about processes exact locations when they migrate.

The Node Service of each node intercepts the outgoing traffic from the node, and redirects it to the CS. (Here it is possible to present a wrapper service for the application service, which should intercept the outgoing messages, but this will complicate the infrastructure, that is why a better solution is this to be a function of the Node Service). After the NS intercepts a message, it adds to it the physical address of the node and forwards it to the CS. The CS delivers it to the destination. When the destination service replies, its NS intercepts the reply and attaches to it its physical address. Thus any further communication does not pass through the CS, but takes place directly between the two services (actually, between their NS). For each message there is a time-out period. The Node Services of the communicating processes keep tables with the corresponding logical and physical address. They use these tables when the traffic goes directly and does not pass through the CS. If there is no reply after the end of the time-out, the corresponding record from the table will be removed. This may happen if the correspondent does not want currently to communicate, if it crashed or finished, or if it migrated. In this case the outgoing traffic should again pass through the CS, because if the process migrated, the CS would know its new location.

In this communication model, services are single-session in the sense that when multiple requests come for a single service multiple instances of the service are started, eventually on different nodes. Thus mobility of the session is automatically provided with the migration of the service – the checkpoint which saves the status of the service actually saves the session too. The checkpoint should also include the user token so that after the service finishes work the result is sent to the right user.

Group formation is supported by the aggregator services that represent a particular functionality [9]. Services in the cluster, once registered, they can join and leave the group on demand, and the indexing components of the platform always remain updated. Group communications are built on top of the mirrored and parallel interfaces that each aggregator service publishes. This collective layer on top of the resources “fabric” layer, allows the traditional semantics of mirrored invocation for higher reliability; and SPMD (Single Program, Multiple Data) for distribution of the load to many nodes in the cluster. The mirrored interfaces can be easily extended to support MPI-like collective operations like Gather, Scatter and synchronization barriers while Broadcast and filtered selective Multicast is already supported.

4.2. Monitoring and Mobility Management

In order to monitor mobility, we are calculating the probability that a device will go off range in the next discreet time slot. We are using the straightforward and realistic assumption that the closer the device is to the access point, the better the chances it will remain in range [8]. Distance in this case is measured in milliseconds as the end-to-end communication delay between the node and the proxy behind the access point. This is in essence a “virtual” distance as it doesn’t depend on the actual distance but on the environmental conditions and obstacles between the two endpoints. In order to measure this distance we can use simple ICMP packets –i.e. ping utility. Then, let S_f denote the network’s standard range, the probability that a device n stays in range can be calculated as:

$$P_n^r = \frac{|S_f - S_n|}{S_f}$$

where S_n is the “virtual” distance of the device from the proxy. S_n can further be enhanced by taking into consideration previous measurements so that we try to “predict” the next distance. Assuming we keep track of the last j distances, S_n can be calculated as follows:

$$S_n = \sum_{k=1}^{j} W_k S_n^{j-k} \quad \text{where} \quad W_k = \frac{k}{C} \quad \text{and} \quad C = \sum_{k=1}^{j} k$$

where W_k is a weighting factor to give more importance to recent measurements.

We are making use of these calculations in order to prioritize the services in the local cluster indexing components. That is, the higher the probability a device will remain in range, the higher the chances it will be selected for an execution before other devices with lower probabilities. Furthermore, when detecting a handoff situation,

Of course the virtual distance is not the only measurement taken into consideration when monitoring the mobile devices: hardware resources like CPU, memory, network and battery should be monitored as well in order to anticipate possible failures due to lack of resources. For this, the intelliagents installed in the devices, periodically provide information on the dynamic state of the host: CPU load, memory load, network load and battery levels. These periodic updates serve also heartbeats to determine “hard” failures – sudden hardware or communication failures that could not be predicted. After a certain timeout limit, the device is considered offline and action needs to be taken. Just like all features of the platform, heartbeats are totally configurable by the administrator. This ensures that different application needs are met in different installations

It is possible every service to have a personal intelliagent, different from the one responsible for monitoring the status of the device. When the monitoring agent detects possibility of failure of the device, migration of all services on this node will be performed to another device in the cluster. On the other hand, if a failure of a single service is detected, which is not due to problems with the device, it will be restarted, either on the same or on another mobile node.

4.3. Checkpointing and Migration Management

Migration of a running task/process may happen in one of the following cases:

- the node, on which the task is allocated, fails and the task is moved to another node (for fault tolerance)
- the task itself fails
- there is a node with less load and the task is migrated to it (for load balancing)

Monitoring Service (MS) keeps track of running services and in case of service execution failure or node failure it informs RMS. RMS should decide what to do – restart the task from the last checkpoint or reallocate the task and restart it from the last check-point or the beginning. When a process is migrated, RMS updates the new location – the previous ProcessID is mapped to a new NodeID and a new ClusterID if the process is reallocated to another cluster. The latter may happen when the task was reallocated from Grid Resource Management Service (GRMS).

5. Integration within the STE Institute

The high-level system communication services supporting different types of communications in grids of non-dedicated or resource limited environments ware not addressed in the roadmap of the STE Institute. However in this paper we addressed the problem of building a generic platform supporting system communications. We consider the clusterification and hierarchical proxies are the key approach for masking the various types of service/task mobility caused by system requirements, application or resource failure.

6. Conclusion and Future Work

Our overview reveals that ProActive groups of objects is very close to our concept of building clusters that support the communication transparency between migrating services as well as on inter-cluster level. Another useful component of such integrated platform is also the Ibis IPL with its support of abstract communication channels. To achieve full mobility transparency these tools are to be upgraded with the system functionalities of our integrated platform such as mobility monitoring, checkpointing, migration and distant launching. A further step in our model is to consider the possibility of different levels of support of persistence – e.g. the communications of the persistent and non-persistent applications as well as communications for the multi-session or multi-instance services.

References

[1] Baude, F., Denis Caromel, Fabrice Huet and Julien Vayssiere. Communicating Mobile Active Objects in Java. *Proceedings of the 8th International Conference on High Performance Computing and Networking Europe*. May 8 - 10, 2000. Amsterdam, The Netherlands.

[2] Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, and Henri E. Bal. Ibis: a flexible and efficient Java based grid programming environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079-1107, June 2005.

[3] Fukuda, M., Y. Tanaka, N. Suruki et. al. A Mobile-Agent-Based PC Grid, *Automatic Computing Workshop Fifth Annual International Workshop on Active Middleware Services (AMS '03)*, 2003, pp. 142-150.

[4] Juhasz, Z., K. Kuntner, M. Magyarody, G. Major and S. Pota, JGrid Design Document, Department of Information Systems, University of Vezsprem, available: http://pds.irt.vien.hu/jgrid/doc/ documentation/ JGrid_Design.pdf

[5] Pota, S., G. Sipos, Z. Juhasz and P. Kacsuk, Parallel Program Execution Support in the JGrid System, *Distributed and Parallel Systems: Cluster and Grid Computing*, *Kluwer International Series in Engineering and Computer Science*, Vol. 777, Budapest, Hungery, 2004.

[6] 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.

[7] Lazar Kirchev, Minko Blyantov, Vasil Georgiev and Kiril Boyanov, A Communication Model Supporting Process Migration in Grid*, presented at EXPGRID, Paris, 21 June 2006*

[8] S. Isaiadis, V. Getov, “Dependability in Hybrid Grid Systems: a Virtual Clusters Approach,” in Proceedings of the IEEE JVA International Symposium on Modern Computing, 2006

[9] S. Isaiadis and V. Getov, "A Lightweight Platform for Integration of Mobile Devices into Pervasive Grids", in Proceedings of High Performance Computing and Communications (HPCC), 2005.

Session II: Integrated Environments

Executing Parameter Study Workflows in the P-GRADE Portal

Gergely Sipos[1], Ariel Goyeneche[2], Tamás Kiss[2], Peter Kacsuk[1]
[1]*MTA SZTAKI Computer and Automation Research Institute*
H-1518 Budapest, P. O. Box 63, Hungary
[2] *Centre for Parallel Computing, University of Westminster*
115 New Cavendish Street, London, W1W 6UW

Abstract. Significant segment of applications running on production grids are parameter studies. Parameter studies realize SIMD style parallelization, namely the same code is executed on several independent parameter sets simultaneously. Although several tools support the parameterized execution of a single job or a single Grid service, no significant research has been carried out on user friendly environments that support the parameter study style execution of Grid workflows. The paper presents our concept for parameter study workflows and the way it has been implemented in the P-GRADE Portal. The P-GRADE Portal now provides an intuitive development and executor environment for parameterized workflows. The executor subsystem of the Portal has been designed to prevent users from flooding production Grids with large number of computational tasks. In order to illustrate the capabilities and benefits of the Portal, a workflow for molecular dynamics analysis using the CHARMM chemistry application is presented. It is shown how the parameter study extensions of the Portal make the creation and execution of this computationally intensive workflow more efficient.

1 Introduction

Parameter studies (sometimes called “parametric studies” or “parameter sweeps”) are applications in which the same binary (job or service) should be executed with a large set of input parameters. Parameter studies can run efficiently on Grids, since these applications consist of large number of independent tasks that can be executed on different resources simultaneously. Indeed, there are several projects that demonstrated that parameter study applications are the optimal way to utilize Grids [1]. However, most of these projects tackled only single job parameter studies. The real challenge is to support the parameter study style execution of complex applications that consist of large number of jobs/services connected into a workflow.

There have been three projects that tried to combine parameter studies with workflow-level support. ILab [2] manages special parameter study oriented workflows. With the help of a sophisticated GUI, ILab users can explicitly define how to distribute and replicate the parameter files in the Grid and can launch independent jobs to process different segments of data files. ILab provides a very static, restricting concept, thus the exploitation of the dynamic nature of Grids cannot be achieved. The SEGL [3] approach puts more emphasis on handling the Grid as a dynamic entity. SEGL also provides a GUI to define workflows and to hide low level details of the underlying Grid. SEGL workflows provide tools for several levels of parameterization, repeated processing, data archiving, handling conclusions and branches during the processing as well as synchronization of parallel branches and processes. The problem with the SEGL GUI is that it might be too sophisticated, requiring far too much skills than Grid end-users have. Furthermore, both ILab and SEGL are connected to one particular Grid, although in case of a parameter study execution exploit as many resources as possible is often required, even if these resources should be collected from different Grids. The result of the third project is the MOTEUR parametric workflow manager tool [5]. MOTEUR is capable of scheduling service workflows described in SCUFL and to execute them over multiple parameter sets. Despite MOTEUR has a quite developed runtime system and provides many options to define parametric workflows, it does not have graphical interface, thus its usage requires programming knowledge.

Our goal was to provide an easy to use graphical environment for the development and execution of workflow based parameter studies. Moreover, the targeted tool should interoperate with every widely used production grid platform and should minimize the load on the connected resources even in case of workflows executed on large parameter spaces. The starting point for our project was the P-GRADE Grid portal [4] that provides a workflow-oriented developer and Grid executor environment integrated into a Web portal. Because P-GRADE Portal interfaces with LCG, gLite, Globus and ARC middleware based Grids, it was a perfect candidate to achieve our goals. Moreover, because a single P-GRADE Portal installation can be connected to multiple Grids at the same time [4], parameter study tasks can be more efficiently distributed by the Portal than using a single Grid environment. In 2 the workflow concept of P-GRADE portal is introduced. This concept has been considered as the starting point for our work. In Section 3 the notion of parameterized workflows is introduced and the cross-product and black box parameterized workflow execution strategies are detailed. In Section 4 the P-GRADE Portal based implementation of the concepts are described. Finally in Section 5 we draw conclusions.

2 The Workflow Concept of P-GRADE Portal

P-GRADE portal consists of two main components: a portal server application built with Web technologies and a workflow editor implemented as a downloadable Java program. The editor is used for workflow development, while the portal server is mainly used to manage the execution of workflows in the connected Grid(s). A P-GRADE Portal workflow is a Directed Acyclic Graph (DAG) where nodes of the graph can be jobs (sequential, MPI or PVM) or application services. Input and output ports can be connected to any node of a workflow. Input ports represent input files to be processed by the node, output ports represent output files to be generated by the node. Input and output ports can be connected by directed arcs (from an output port to an input port). Arcs represent the file transfer operations between nodes that must be resolved by the workflow runtime system of the portal during workflow execution. Indeed, the run-time system takes care of transferring and renaming the input/output files, so the user is completely released of job and file management during workflow execution.

Based on this semantics two-level parallelism can be easily achieved: the first level is among parallel branches of the workflow (inter-node parallelism), the second level (intra-node parallelism) is within a workflow node, if the job or service represented by the node is a parallel code (e.g. MPI or PVM). Our goal with the parameter study extension was to extend it with a third level, namely to enable the execution of the same workflow on different input data sets simultaneously. This third level represents SIMD (Single Instruction Multiple Data) style parallelization.

3 Semantics of Workflow-Level Parameter Studies

Turning a workflow into a parameter study means that its input data set is parameterized. Because input files are the main inputs of P-GRADE Portal workflows, we decided to support the parameterization of file inputs. As input files are represented by input ports in the workflow editor, this practically means that a new input port type should be introduced: PS input ports. According to our definition a “PS input port” is used to send a set of parameter files to a workflow. If there is a PS input port connected to a workflow, then the workflow should be executed as many times as the number of parameter files sent to the workflow through the port. From now on if a workflow (WF) has at least one PS input port connected to it is called a PS workflow (PS-WF). Workflows can be easily parameterized according to this rule if there is only one PS input port in a PS-WF. However, if more than one PS input ports are connected to the graph there are several options to do parameterization. As it has been shown in [5] in such cases files coming from different PS input ports can be combined in several ways. In our current work we focus only onto the “cross-product” combination of input files, because this combination type is the easiest to understand and to use by the users.

Cross-product of input files means that the files sent to a workflow through PS input ports must be combined in every possible way. If there are *m* PS input ports connected to a PS-WF, then the cross product operation defines an *m* dimension parameter space. If N_i denotes the number of input files to be sent through the i. PS input port, then the cross product defines $M = N_1 \, x \, N_2 \, x \ldots x \, N_m$ points in the parameter space. Consequently, *M* workflow executions will be generated from the PS-WF. (One workflow execution for each point of the parameter space.) A PS-WF executed for one point of the parameter space is from now on referred to as an “executable workflow” (e-WF).

The simplest approach to generate the *M* e-WFs from the PS-WF is by the “black box” strategy. This strategy means that the workflow graph is considered as a black box and executed once for each point of the parameter space, without taking care of the inner structure of box. The runtime system of the portal generates *M* instances from the workflow and extends each instance with the input file set determined by one point of the parameter space. At the end all the e-WF can be executed in the Grid.

Although the black box semantics results a simple and easy to implement PS-WF executor subsystem, it has a drawback, namely that is does not minimize the number of computational tasks. Every job/service of the original workflow is executed for every point of the parameter space, even if there are multiple jobs with the same input files started. Such situation happens when a PS input port is connected to one of the *non-root* node of the workflow graph. If N_j denotes the number of parameter files to be sent to the non-root node through this PS input port, then every parent node of this non-root node will be executed N_j times with the same input data. Obviously one execution would be enough. Because the black box strategy does not transfer data from one e-WF to another, it cannot discover such extra executions. We are already working on the elaboration of a more intelligent execution mechanism [6].

4 Implementation of the Concepts in the P-GRADE Portal

From the users points of view the lifecycle of PS-WFs consist of three main phases: 1. development of the WF that will be later executed on a large parameter space; 2. Defining the parameter space, thus transforming the WF into a PS-WF; 3. Submitting and monitoring the execution of the PS-WF in the Grid. Because the first phase was already covered by the P-GRADE Portal [4] we do not go into further details on that here. Phases 2 and 3 represent new stages in the application development, thus new tools had to be developed and integrated into the Portal. These tools are introduced in the remaining part of this section.

4.1 Turning Workflows into PS-WFs

To support the parameterization of WFs the workflow editor component of the P-GRADE Portal had to be slightly extended. With the new editor the user can open an existing workflow and can transform it into a PS-WF by turning any of its input ports into a PS input port. (Remember, that a PS input port defines one dimension of the parameter space.) A PS input port represents *n* parameter input files to be processed by the workflow. These parameter input files will be given to different e-WFs during the execution phase.

PS input ports can be defined on so called property windows in the workflow editor. On each PS port property window a directory can be defined for the corresponding PS input port. This directory must contain the parameter input files that should be sent to the workflow through the port. In the current implementation the directory must refer to a Grid directory from one of the connected Grids of the portal. The user should place the series of parameter input files into this directory before submitting the PS-WF into the Grid. Because parametric input data is stored on storage resources instead of the portal server, the concept provides a scalable solution even for large parameter spaces. However, not only the input data but also the output data can fill a large space: one instance of every output file of the workflow is created by every e-WF. The portal should be released from storing this huge mass of data, result files of PS-WFs should be stored on Grid storages too. Obviously result files stored on the portal server would make the disc of the server a serious bottleneck during PS-WF execution. Our solution for the problem is the following: During the execution of an e-WF the portal server caches where output file generated by the e-WF. After the e-WF has been finished, its result files are archived into a single file and moved into a pre defined Grid directory. At the end of the whole PS-WF execution this Grid directory contains every result file belonging to the experiment. The Grid directory must be defined for the PS-WF in advance of workflow submission by the user.

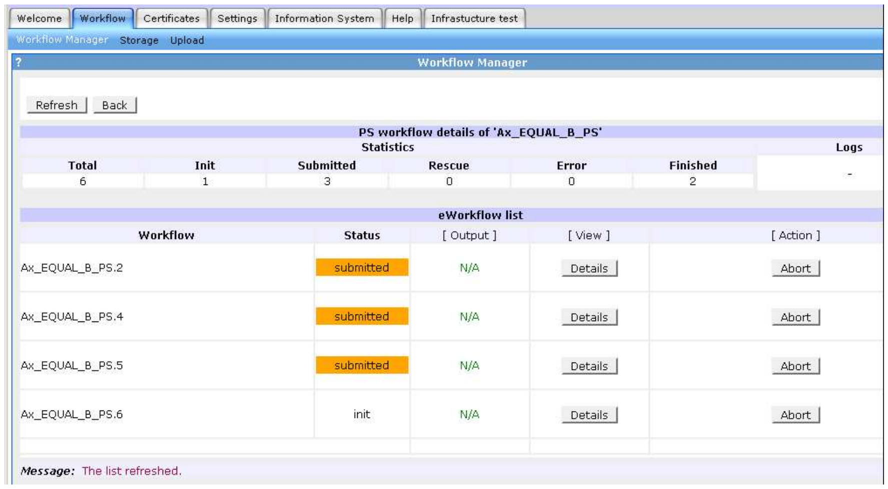
As a summary we can say that turning an existing WF into a PS-WF is an extremely easy task. Simply turn some of the input ports to PS input ports and define a Grid directory where the result files to be generated by the e-WFs should be stored.

4.2 Executing and Monitoring PS-WFs

Monitoring becomes extremely important when a Grid application consists of thousands of workflows, each of them containing hundreds of jobs or services. The challenge is to visualize the execution statuses of so many e-WFs and their nodes in an easily understandable and manageable way. Facilities provided by the P-GRADE Portal for normal workflows were good starting points for us to define GUI elements for PS-WF execution and monitoring. The Workflow portlet, which is used for the management of normal workflows, has been extended with a new section. This new section provides monitoring facilities for PS-WFs. In the PS-WF section the user can submit, open and delete PS-WFs and can follow their progress during execution. A detailed view can be obtained for any PS-WF. In detailed mode the portal graphically presents the statuses of the currently active e-WFs of the PS-WF (e.g. the running e-WFs). Moreover, for each e-WF the user can get an inner view, which means that the statuses of the jobs of the active e-WF can be listed. In Figure 1 the detailed view of a PS-WF is presented.

As it can be seen in Figure 1, the PS-WF has 4 active e-WF. In the upper part of the figure the total number of e-WFs generated from this PS-WF can be seen: 6. Although the PS-WF resulted 6 e-WFs the portal did not start all these 6 e-WFs simultaneously. This is because a limit has been integrated into the portal for the number of parallel e-WF execution. The limit protects the connected Grids from being flooded, as it maximizes the load the portal can put on the Grid.

Fault-tolerance in PS-WF execution has an outstanding importance, since a PS-WF typically uses many Grid resources for a long period of time. Fault tolerance for normal workflow execution is provided by the Portal at several levels. First of all, any job of a WF can be assigned to a Grid broker. The broker will resubmit the job if its execution fails for any reason on the selected resource. If a job assigned by the user to a particular resource fails, or the broker reached the resubmission limit then the WF enters into “RESCUE” status. The user can reallocate the failed job to a different Grid or onto a different resource and the execution can be resumed from the point of failure. Because according to the black box strategy a PS-WF is degraded into a set of e-WFs that can be executed as they were normal workflows, services provided by the Poral for workflow fault tolerance are available for e-WFs too.



PS workflow details of 'Ax_EQUAL_B_PS'

Statistics

Total	Init	Submitted	Rescue	Error	Finished	Logs
6	1	3	0	0	2	-

eWorkflow list

Workflow	Status	[Output]	[View]	[Action]
Ax_EQUAL_B_PS.2	submitted	N/A	Details	Abort
Ax_EQUAL_B_PS.4	submitted	N/A	Details	Abort
Ax_EQUAL_B_PS.5	submitted	N/A	Details	Abort
Ax_EQUAL_B_PS.6	init	N/A	Details	Abort

Message: The list refreshed.

Figure 1. Detailed view of a PS-WF in the P-GRADE Portal

5 Molecular Dynamics Simulation by a Parameter Study Workflow

CHARMM is a software application for modeling the structure and dynamics of molecular systems [7]. Systems ranging from individual organic molecules to assemblies of large biological molecules can be modeled with CHARMM. Because CHARMM possesses a wide variety of methods and routines, it can be used to address a wide range of problems. A classical problem CHARMM is used for is studying proteins in molecular dynamics (MD) simulations based on empirical force field potentials. A typical MD simulation consists of four parts: 1. heating up of the system to a desired temperature, 2. equilibration of the system at a desired temperature, 3. production phase in which a coordinate trajectory is generated 4. analysis phase when the trajectory with various tools available in CHARMM are analyzed.

The manual management of files and simulation parameters of MD experiments often result in an error. Therefore an automated system, in which all phases of one simulation are performed without interruption in the form of a workflow, is a crucial element of successful run. More importantly, from a practical point of view, submission and monitoring of a large number of computer jobs is a very time-consuming activity that can be by far improved by parameter study techniques.

Multiple MD simulations running in parallel, preferably on hundreds of input data sets, may be needed to observe even a single event. In order to improve the management of submission and monitoring of the large number of production and analysis faces, the PS extension of the P-GRADE Portal has been used. Based on the PS P-GRADE Portal the time required to create and execute the experiment has been significantly shortened compared to command line techniques or the development of an application specific environment. On top of that, a simple modification in the seeds generator offered us a complete new set of parallel workflows ready to be submitted. The PS P-GRADE Portal based molecular dynamics simulation workflow has been demonstrated at the Supercomputing 2006 Exhibition in the USA.

6 Integration Within the STE Institute

The P-GRADE Portal is serving the users of the largest production Grids all around the world. The parameter study extension of the Portal provides solution for serious problems these users are faced with. The achieved results thus represent a significant link between Coregrid and Grid end users representing research and industry. Condor DAGMan enactment engine is used by the P-GRADE Portal for workflow management purposes. DAGMan has been extended by our group with several middleware specific scripts to make job submission and file management possible from the Portal into Globus, LCG, gLite and ARC middleware based Grids. These scripts act as gateways between the users' applications and standard production level middleware services. Coregrid task 7.3 is working on a high level toolkit, which will establish an abstract layer on top of component based Grid frameworks, primarily on top of the component framework and mediator components under development by Coregrid tasks 7.1 and 7.2. In the long run this toolkit should take the role of mediation inside the Portal. However, this integration is reasonable only if we do not loose any of the existing portal features, thus if the toolkit is able to interoperate not only with Coregrid specific, but also with the widely used production Grid services. In this way the collective result of Coregrid WP7 would represent an important link towards projects and consortiums working on component based middleware systems.

7 Conclusions

In the paper the concept and implementation of a workflow based parameter study developer and executor environment has been presented. Although the currently used black box strategy does not minimize the number of computational tasks of PS-WFs, the built-in limitation for the number of concurrently running e-WFs prevent users from flooding the connected Grids with large number of computational tasks. We are already working on a more sophisticated PS-WF executor mechanism [6] and on a general meta-broker which can submit jobs into different production Grids. While the former improvement would minimize the number of computational tasks generated by parameter studies, the meta-broker would result better utilization of Grid resources participating in PS-WF execution.

8 References

[1] Abramson, D., Giddy, J., and Kotler, L., High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?, IPDPS'2000, Mexico, IEEE CS Press, USA, 2000.

[2] McCann, K. M., Yarrow, M., deVivo, A. and Mehrotra P., ScyFlow: An Environment for the Visual Specification and Execution of Scientific Workflows, GGF10 Workshop on Workflow in Grid Systems, Berlin, 2004.

[3] N. Currle-Linde, F. Boes, P. Lindner, J. Pleiss and M.M. Resch, A Management System for Complex Parameter Studies and Experiments in Grid Computing, in: Proc. of the 16th IASTED Intl. Conf. on PDCS (ed.: T. Gonzales), Acta Press, 2004.

[4] P. Kacsuk and G. Sipos, Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal, Journal of Grid Computing, Vol. 3, No. 3-4, pp. 221-238, 2005.

[5] Tristan Glatard, Johan Montagnat, Xavier Pennec, Medical image registration algorithms assesment: Bronze Standard application enactment on grids using the MOTEUR workflow engine. in Proc. of the HealthGrid conference 2006, pages 93-103, Valencia, Spain, 2006.

[6] P. Kacsuk, Z. Farkas, G. Sipos, A. Tóth, G. Hermann, Workflow-level Parameter Study Management in multi-Grid environments by the P-GRADE Grid portal, in Proc. Of Second International Workshop on Grid Computing Environments (GCE'06), Tampa, Florida, USA, November, 2006.

[7] CHARMM Development Project: http://www.charmm.org/

Legacy Code Repository with Broker-based Job Execution

Gabor Kecskemeti[1,3], Gabor Terstyanszky[2,4], Tamas Kiss[2,3], Peter Kacsuk[1,3],

[1]*MTA SZTAKI Computer and Automation Research Institute*
H-1518 Budapest, P. O. Box 63, Hungary
[2] *Centre for Parallel Computing, University of Westminster*
115 New Cavendish Street, London, W1W 6UW
[3]*CoreGrid Institute on Grid Systems, Tools and Environments*
[4]*CoreGrid Institute on Grid Information, Resource and Workflow Monitoring Services*

Abstract

As Grid technology matures more and more production Grids become available to run computationally intensive scientific applications. However, as these Grids are based on different middleware solutions interoperation between these platforms is one of the major challenges the Grid community is facing today. The P-GRADE/GEMLCA portal is a workflow oriented Grid portal that supports the execution of workflow components simultaneously on multiple Grids based on different underlying technology. Jobs can be submitted to Globus or LCG/gLite based Grids, or can be selected from the GEMLCA legacy code repository. However, as GEMLCA is implemented on top of GT4, it was capable to interact only with GT2 and GT4 based Grids until recently. Moreover, legacy codes selected from the GEMLCA repository were statically mapped to resources. This paper describes how GEMLCA has been ported to the LCG/g-Lite based EGEE infrastructure. Besides simply porting GEMLCA to another middleware it had to be made capable to interact with the EGEE broker solution. A legacy code can not only be selected from the repository, but using its legacy code interface description it is also defined which resources are capable to execute the given code. Based on this information the broker can find the most suitable resource at workflow execution. As a result of this integration the P-GRADE/GEMLCA portal is capable to interact with both Globus and LCG/g-Lite based Grids by the means of either direct or broker-based job submission, or by browsing and selecting the executable from the legacy code repository. The described work illustrates how two important components of production Grids, a legacy code solution and a Grid broker can be successfully integrated.

1. Introduction

The Grid computing environment requires special Grid enabled applications capable of utilising the underlying Grid middleware and infrastructure. As the Grid becomes stable and commonplace in both scientific and industrial settings, a demand is created for porting a vast legacy of applications onto the new platform. Companies and institutions can ill afford to throw such applications away for the sake of a new technology, and there is a clear business imperative for them to be migrated onto the Grid with the least possible effort and cost.

Grid computing is now progressing to a point where reliable Grid middleware and higher level tools will be offered to support the creation of production level Grids. A high-level Grid toolkit should definitely include components for turning legacy applications into Grid services. The Grid Execution Management for Legacy Code Applications (GEMLCA) [1] enables legacy code programs written in any source language (Fortran, C, Java, etc.) to be easily accessed through a Grid Service interface without significant user effort. GEMLCA does not require any modification of, or even access to, the original source code. A user-level understanding, describing the necessary input and output parameters and environmental values such as the number of processors or the job manager required, is all that is needed to port the legacy application binary onto the Grid. Once the interface is described GEMLCA legacy codes are also available for other authorised users to run them with custom parameters or to include them in their Grid workflows.

GEMLCA was originally implemented to support service oriented Grid middleware, namely Globus toolkit version 4 (GT4) [2]. GEMLCA was implemented as a set of GT4 Grid services that submit legacy code jobs to local job managers like Condor or PBS using the GT4 WS-GRAM service. However, it became apparent that different production Grids currently use different Grid middleware without providing interoperability between them. In order to offer a legacy code converter tool, like GEMLCA, that supports the large variety of existing Grid middleware solutions, a more flexible approach was required. GEMLCA was re-engineered to be able to utilise different back-end plug-ins submitting to different Grid middleware.

The original GEMLCA concept only supported direct mapping of legacy code jobs to resources. GEMLCA legacy codes were tightly bound to a particular resource. At job execution or workflow creation the user first selected the target resource hosting the executable legacy code and then defined custom parameters. As more and more production Grids are experimenting with resource brokers it was necessary to extend the GEMLCA concept in order to interact with these brokers.

This paper describes how GEMLCA is capable to support multiple Grid platforms by creating middleware specific back-ends. These back-ends can also interact with existing resource brokers, like the LCG/g-LITE broker of the EGEE Grid.

2. Legacy Code Support for Multiple Grid Middleware

GEMLCA represents a general architecture for deploying legacy applications as Grid services without re-engineering the code or even requiring access to the source files. The deployment of a new legacy code service requires only a user-level understanding of the legacy application, i.e., to know what the parameters of the legacy code are and what kind of environment is needed to run the code (e.g. multiprocessor environment with ‘n’ processors). The execution environment and the parameter set for the legacy application is described in an XML-based Legacy Code Interface Description (LCID) file that should be stored in a pre-defined location. This file is used by the GEMLCA Resource layer to handle the legacy application as a Grid service.

GEMLCA is integrated with the workflow-oriented P-GRADE Grid portal [8]. The P-GRADE portal enables the graphical development of workflows consisting of various types of executable components (sequential, MPI or PVM programs), execution of these workflows in Globus-based Grids relying on user credentials, and finally the analysis of the correctness and performance of applications by the built-in visualization facilities. The P-Grade portal has also been extended with a GEMLCA administration portlet. This portlet hides the syntax and structure of the LCID file from users so that users do not have to know LCID specific details, and do not have to be familiar with possible modifications in legacy code description whenever a new GEMLCA release would require it. The user has to specify exactly the same parameters as in the XML file but this time using a simple Web form. The LCID file is created automatically and uploaded by the portal to the appropriate directory of the GEMLCA service.

GEMLCA has been internally designed in three layers. Each of these layers simulates an encapsulated black-box that is committed to deliver a well-defined functionality to the layer above that, independently of the underlying Grid middleware solution.

The first, *Front End Layer*, offers a set of functionalities as Grid Services. Any authorised Grid client can utilise the functionalities to deploy and use legacy code programs on a GEMLCA Resource. The GEMLCA functionalities are expressed with the help of the following GEMLCA Grid services:

- *GLCAdmin:* After authentication, *GLCAdmin* enables the client to modify the XML-based LCID file of an already deployed legacy code, or to create a new LCID file and upload it to the GEMLCA Resource.
- *GLCList:* Returns a list of already deployed legacy codes.
- *GLCProcess: S*ubmits the legacy application using GEMLCA to a compute server and gets job status and results back.

The GEMLCA architecture manages the concepts of legacy code processes (*LCProcess*). GEMLCA manages these processes, and they do not have to be confused with ordinary Grid processes. The *Core Layer* is in charge of the management of these internal concepts. When a GEMLCA Grid Service is created, automatically a *LCProcess* object is constructed within it. The *LCProcess* contains the memory structure to handle the legacy code input parameters and manage a unique hard disk environment to store legacy code input and output files. When a *LCProcess* is submitted, a *GLCEnvironment* object is created with new memory and hard disk structures. Using these concepts, several *GLCEnvironments* can be created, submitted and destroyed from a single *LCProcess*. Additionally, it allows multiple submissions in a multi-user environment, where user-specific information, input and output files and parameters, can be preserved separately from other instances running on the same node.

The final *Back End Layer,* is connected to the Grid middleware on the host where the

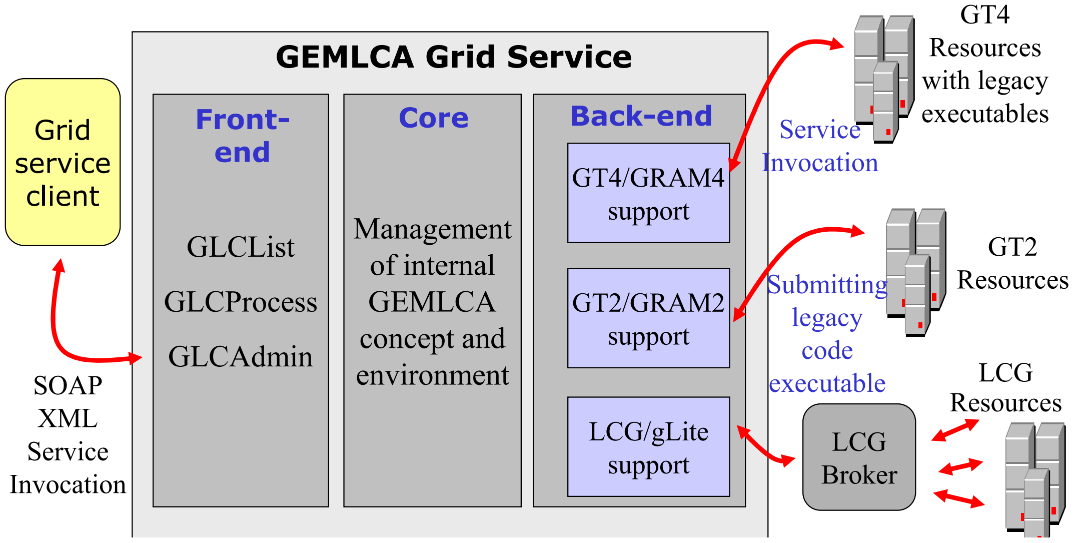


Figure 1 The three layered GEMLCA architecture with multiple back-ends

architecture is being deployed. It knows the different ways to contact, submit jobs, and get status back from the correlated middleware. This layer, as it was described in the previous section, was originally tightly coupled with the GT4 Grid middleware. However, this feature restricted the deployment of the GEMLCA architecture to GT4 based Grids only. As the largest production Grids currently use either GT2, like the UK National Grid

Service [3] or the Open Science Grid in the US [4], or LCG/g-Lite [5] like the EGEE Grid [6] it was inevitable to redesign the GEMLCA back-end layer to support multiple Grid middleware solution.

Figure 1 shows the three layer GEMLCA architecture with multiple back-ends. Besides the original GT4 support described in [1] GEMLCA has been extended to submit to GT2-based Grids. As detailed in [7], the GT2 GEMLCA back-end allows users to create legacy code repositories on the GEMLCA resource, and select and submit the codes to the remote GT2 gatekeeper utilising Condor-G. This paper concentrates on the design principles of the third back-end plug-in that interfaces with LCG/g-Lite based Grids through the EGEE broker.

3. GEMLCA in broker-based Grids

Production P-GRADE Grid portals, installed in the EGEE Grid, submit and execute jobs on EGEE resources using the LCG broker. GEMLCA has to be extended towards the EGEE platform to run legacy code applications on the EGEE Grid using the LCG broker. In order to run legacy code applications on EGEE resources with the LCG broker the following requirements should be satisfied:

1. GEMLCA should be able to submit jobs through the LCG broker using its JDL language instead of using the Globus's RSL.
2. GEMLCA should be able to send input files and retrieve output files using the LCG broker.
3. The LCG broker should submit legacy code applications only those resources where they can be executed taking into consideration their specific configuration and execution requirements.
4. The LCG broker based job submission for legacy code applications should be integrated seamlessly with the P-Grade Grid portal to minimise user efforts to learn and apply this solution.

The GEMLCA team created a legacy code repository, called GEMLCA-R, to run legacy code applications as jobs on the Grid. The repository contains the binary and may include some input files of the legacy code applications. Information about these applications can be retrieved through the repository list. Code owners first, should check whether their applications can run on a resource next, they should upload the applications into the GEMLCA-R, assigned to the resource. After uploading legacy applications users with valid certificates and the required authorisation can select and execute legacy codes using a list of available legacy code applications registered with a GEMLCA-R.

We identified two approaches to submit legacy code applications through the LCG broker:

1. One-to-one brokering
2. One-to-many brokering

3.1. One-to-one brokering

According to this approach a legacy code application, implemented as a GEMLCA Service, is assigned to one particular EGEE resource. To describe this assignment the service should extend the JDL file to restrict the submission of the legacy code to the allocated resource. As a result, when GEMLCA submits the legacy code application to the LCG broker, it forwards the code to the resource, which is specified in the JDL file. In this approach legacy codes could be assigned to resources using either the unbalanced or balanced job submission.

Unbalanced job submission. Each EGEE resource should have its own repository, which is maintained by GEMLCA services mapped to the resource. The mapping between the resource and the service is defined by the code owner and done by the P-GRADE portal. Submitting a legacy code as a GEMLCA service to an EGEE resource requires an extra input file, which defines the EGEE resource where the execution should take place. Using the unbalanced job submission, the user selects an EGEE resource specifying a GEMLCA-R. The portal forwards the job to the LCG broker, which is a single point of entry to the EGEE Grid. The broker sends the job to the resource defined by the user not performing any kind of brokering activities.

Balanced job submission. It is the extension of the unbalanced job submission. The P-GRADE portal has its own built-in broker service, which can interface with GT2 and LCG Grids. The built-in broker selects the EGEE resources for the GEMLCA service using an aggregated list, which contains all legacy codes registered with GEMLCA services. The aggregated list leads to longer job setup times because of queries made to create the list and caching the legacy code list of GEMLCA services. Using the built-in broker, no resource selection UI is available. In this approach the user forwards the job to the built-in broker that selects an EGEE resource specifying a GEMLCA-R. The portal forwards the job to the LCG broker, which sends the job to the EGEE resource selected by the built-in broker.

3.2. One-to-many brokering

This approach assumes that the list of EGEE resources where a legacy code can be executed has been compiled and is available. The GEMLCA service, representing the legacy code, has to add this list to the JDL file. Having this list the LCG broker is able to select one of the resources, which is capable to run the legacy code. The user interface of the portal with broker is not different from the user interface without broker.

3.4 JDL file generation

The JDL file can be generated by either the Grid portal or GEMLCA. If the portal creates the JDL file, the GEMLCA does not even have to understand the contents of the JDL file. In this case GEMLCA acts as a “primitive” job submitter, e.g. GEMLCA argument integrity check is not used. To avoid this situation, the GEMLCA service generates the JDL file. This can be done in two ways: complete and partial JDL generation.

Complete JDL generation. The GEMLCA service has a full control over the contents of the JDL file and it provides all the information for the JDL file. To achieve it the legacy code descriptor has to be extended to support the JDL file generation. At each submission the GEMLCA service has to generate the JDL file for the broker. Therefore, there is no need for a JDL parser on the service side. The portal has a JDL writer engine, which could be used for the JDL generation. The drawback of this solution is the user has to be familiar with some EGEE specific details.

Partial JDL generation. GEMLCA creates only those JDL parts, which are different at submissions. This approach is based on a JDL file with a sample submission description. This file can be customised according to the actual state of the legacy code environment (e.g. arguments, inputs). The customization means that the GEMLCA service has to parse the JDL document and write a new one containing the details of the execution. It is quite important that the original JDL might hold some information on resource requirements, and also some job manager and execution specific data (like MPI execution preparation or queue setup), which are the same for all resources. If the generated and the sample JDL files are very similar, the job submission based on these files will be very similar. It happens if the GEMLCA service has to update only the resource restrictions but not the others.

The only question is where to get the sample JDL from. The administrator of the legacy code should pass it with the legacy code description. The P-GRADE portal may offer the generated and submitted JDLs for the GEMLCA or as an extra option the administration portlet should upload the JDL file as an attachment to the description.

4. Integration Within the STE Institute

GEMLCA is serving the users of the largest production Grids all around the world. Extending GEMLCA with multiple back-ends supporting different Grid middleware solutions opened the way to install GEMLCA on different Grid infrastructures serving a much larger user community. The achieved results thus represent a significant link between Coregrid and Grid end users representing research and industry. Within Coregrid task 7.4 researchers identified the challenges and solutions to deploy legacy applications in a Grid environment. Given the importance of legacy applications as described in section 1 of this paper, a component based Grid platform have to include solutions to wrap and present these applications on the Grid. GEMLCA is the most widely used solution to fulfil this task and is now capable to support multiple and different Grid platforms and Grid brokers, as described in this paper.

References

[1] T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, P. Kacsuk: GEMLCA: Running Legacy Code Applications as Grid Services, “Journal of Grid Computing” Vol. 3. No. 1.

[2] Globus Team, Globus Toolkit 4.0 Release Manuals, http://www.globus.org/toolkit/docs/4.0/

[3] The UK National Grid Service Website, http://www.ngs.ac.uk/

[4] The Open Science Grid Website, http://www.opensciencegrid.org/

[5] The gLite website, http://glite.web.cern.ch/glite/

[6] The EGEE web page, http://public.eu-egee.org/

[7] T. Kiss, G. Terstyanszky, G. Kecskemeti, Sz. Illes, T. Delaittre, S. Winter, P. Kacsuk, G. Sipos: Legacy Code Support for Production Grids, Conf. Proc. of the Grid 2005 - 6th IEEE/ACM International Workshop on Grid Computing November 13-14, 2005, Seattle, Washington, USA

[8] P. Kacsuk and G. Sipos: Multi-Grid, Multi-User Workflows in the P-GRADE Grid Portal, Journal of Grid Computing Vol. 3. No. 3-4., 2005, Springer, 1570-7873, pp 221-238

Information Models for Lightweight Grid Platforms

Georgi Pashov[1], Kalinka Kaloyanova[1], Kiril Boyanov[2]

[1]*Faculty of Mathematics and Informatics, Sofia University "St. Climent Ochridsky"*
[2]*Institute on Parallel Processing – Bulgarian Academy of Sciences*
e-mail: [gpashov, kkaloyanova]@fmi.uni-sofia.bg; [boyanov]@acad.bg

Abstract

The choice of suitable information model is a key step to the successful deployment and use of the Grid infrastructure. It has to meet strong requirements for scalability, robustness and performance in the heterogeneous and dynamic Grid environment. Choosing relevant information technology and data model for lightweight grid is even more complicated because of the requirements for simplicity, good functionality and efficiency.

In this paper we review hierarchical and relational approaches for Grid Information Services and special features enforced by the lightweight grid principles. We present the main challenges that have to be taken into account: centralized vs. decentralized architecture, platform's heterogeneity, data distribution and replication, designing functionally complete Grid entity set (e.g. hosts, storage, links, users and services), etc.

1 Introduction

Grid Information Service (GIS) is a complex architecture that maintains structural, uniform and meaningful information about services, resources, participants, etc. in the heterogeneous and dynamic Grid environment, in order to support the discovery, selection and optimization of its components. As the scale and diversity of Grid infrastructure are growing, the information needed for efficient management becomes critical.

GIS is closely connected to the other Grid services. Job execution in grid environment relies on timely access to accurate and up-to-date information for resources and services, spread over wide area. Resource Management Service must be able to locate suitable operational site (OS, CPU, storage capacity, data sources) as well as to examine current and expected system state (CPU utilization, available storage space, etc.) in order to achieve efficient job scheduling. It is important GIS to provide regular information about job progress too. Security service also interacts with GIS in order to authenticate and authorize users. Another use case is a Grid portal, whose functionality is based on the information gathering by GIS.

When an abnormal situation happens, the recent information could be very useful in the subsequent analysis for reconstruction of the events that have led to the failure.

The collected data may be used to optimize the system behaviour according to the conditions on the fabric.

2 Information models in grid

The information model integrates in harmony proper technologies and data structures in order to create a robust GIS. The choice of suitable information model is a key step to the successful deployment and use of a Grid infrastructure.

Below are listed some general questions that elucidate GIS data design features. In the next section we present the GIS technological overview.

2.1 Requirements to the data provided by GIS

A GIS provides information about valuable data objects (in the sense that they bring some benefits for participants). Entities in GIS, whose real-world representation are the objects, enforce clear and standard view on the heterogeneous grid resources. They must meet the following criteria:

- Usefulness – the entity includes only the valuable attributes (only meaningful information)
- Uniqueness – the entities are distinguishable from each other (the user can choose one particular entity among the others of the same type)
- Persistence – the entity is long lasting (transient data, arised accidentally or extremely rarely, are not candidates for entities)
- Generality – the entity is valuable to multiple applications or users

Entities have to be capable of joining to other entities or belonging to more than one category as their functionality requires that. Hence the data model must support complex relationships between entities.

2.2 Kind of data that GIS must support

GIS must ensure functionally complete Grid entity set. That means to be present the whole variety of entities which comply with the above outlined data model. For instance:

- Static Host Information (operating system, version, processors, hostname, etc.)
- Dynamic Host Information (load average, CPU usage, running processes, etc.)
- Network Information (topology, bandwidth, latency, etc.)
- Storage System Information (total disk space, available disk space, etc.)
- Applications (software, data, active services, etc.)
- Users (users, credentials, roles, etc.)
- Organizations (name, security policies, etc.)
- Instruments (radars, etc.)

2.3 Some considerations about data's dynamics

Almost all of the data are interesting as dynamic view (which shows how they were changed over time). Such statistics could be used for: billing in economic grid; system crash investigations; host load forecasting and so on. As a result, supporting tracks for data objects' changes is a good practice. According to the dynamics, there are discrete value's data (last state, historical records for past period) and continuous data streams (that reflect resource's dynamics on fly). Despite of the different ways of using data, however, it is desirable to have a uniform representation that includes mandatory timestamp attribute.

3 Challenges that must be taken into account

Except for data model, the investigation of underlying characteristics, technologies and approaches is a key step to successful deployment and efficient use of the Grid infrastructure. There are several aspects that have to be examined in order to acquire a clear notion of GIS architecture.

3.1 Heterogeneity

A grid environment is a highly heterogeneous. It is very important GIS to be platform independent (built on Java platform for instance) which increases service flexibility and portability and decreases system support efforts.

3.2 Centralized vs. decentralized approaches

A centralized approach (maintaining single data repository) is very easy to manage. Provider-centric GIS ensures that the platform is freed from complex data modelling and binding issues but it introduces a lot of problems, such as:

- Performance: It could be a performance bottleneck if all resources direct their data streams to a centralized information server.
- Scalability: When the number of clients grows, a single information centre may have inadequate capacity to process all the requests.
- Fault tolerance: When the centralized server is not active the whole system would stop. It will be restored only after the information server is on line.

A decentralized approach (distributed repositories), on the other hand, is much more difficult to manage, involves communication and synchronization overheads but ensures more reliable and flexible background for a GIS. The considerations mentioned above are no longer obstacles to good GIS realization. But there are some others:

- Distributed Search: In a decentralized environment meta-data is spread over a large area. In order to obtain relevant information a number of repositories must be accessed. Distributed search means retrieving data from spread information centres and combining results in fast and efficient way. Performance significantly can be improved through indexes which collect and aggregate frequently used data.

Any topologies are possible but a combination of the two strategies sounds more feasible for grid environment. We accept that the greatest part of the activities is in the bounds of administrative domains (logical cluster) and intercommunications are less. So, centralized information schema on local level and distributed peer-to-peer infrastructure that provides global view is the most reliable GIS architecture.

3.3 Replication and caching

In the context of data grid technologies, replication is mostly used to reduce access latency and bandwidth consumption. The other advantages of replication are that it helps in load balancing and that it improves availability by creating multiple copies of the same data. Obtaining precise information on time is vital characteristic of GIS and replication and caching are extremely good strategies for achieving this goal. Frequently needed data can be locally cached and reused. An important question is how stale the information is. A mechanism is necessary to guarantee data freshness. Providing Time To Live (TTL) attribute, indicating how much time data can be used, is a good choice.

However, when a multiple GIS schemas' copies are available how is data consistency provided? Static meta-data is less sensitive to changes and isolated synchronization delays are not so critical. However, latencies in dynamic data synchronization result in returning outdate information, which in turn could lead to inefficient scheduling decisions.

3.4 Security

In a Grid environment security is a complex problem. In order to address robust authentication and communication protection demands a suitable security infrastructure should be built. Adoption of too severe security policy could become performance problem. In general, GIS schema contains resource characteristics and statuses of current and completed jobs, which are not strictly confidential. So, balance between reliable security level and adequate performance should be found.

3.5 Open standards and open source tools

Applying open standards and protocols and well known tools has as consequence improved scalability and better tools support as well as sharing development experience. The fundamental direction for building a flexible and reliable GIS infrastructure should be: open standards, open source tools and portable implementations (based on Java). Whenever open standard or tools agree with grid project specifications it is better to choose them.

4 GIS in lightweight grid context

There are three main tendencies towards Grid infrastructure's functional scope and complexity: integrated grid systems, application-driven systems and middleware. The first two categories are dedicated mostly to particular scientific tasks or business solutions. Middleware is a more generic platform. It facilities grid constructing by providing an abstract layer: a set of loosely coupled basic services that in turn can be used to implement higher-level components and applications.

Lightweight grid, which is a middleware subcategory, provides base functionality and ensures integrated, secure, efficient, adaptable and scalable grid environment. Here, challenges arise from the high diversity and extreme dynamics of the lightweight grid environment, with its non-dedicated and uncertain resources. As a result, the data model must propose uniform information view, optimized strategies for data management, great scalability and so on.

Typical features of lightweight grids are simplicity (in sense of developing, deploying and maintaining) and generality (in sense that they should cover the most of user needs). Also they must be self-installable and self-tuneable without (or with minimal) human interference in their processing.

Below are specified some consideration about lightweight GIS.

4.1 Simplicity

One of the main characteristics of lightweight grid is its simplicity. That means clear information model and easy infrastructure support with minimal number of other installation dependencies. However, simplicity does not mean restricted functionality. Just the opposite, lightweight GIS must provide functional completeness (i.e. data schema including all necessary information so that users may run their tasks without any functional restriction).

4.2 Fast development

Using proven technologies and third party software implementations (databases, LDAP servers, etc.) could reduce development time and project cost as well as facilitates product support. The question is, however, how far these decisions satisfy lightweight GIS requirements. Most of the database systems, for example, are not platform independent. Consequently, it is necessary to support a variety of platforms i.e. difficult installation and maintenance. That, however, discredits the fundamental ideas for simplicity and easy support.

4.3 Fault tolerance

Decentralized GIS schema protects the Grid from total crash in case of failure of some its module. However, in order to ensure more reliable and fault tolerant environment, a backup GIS instance must be available. A primary GIS instance store data in memory or on disc and synchronizes data with standby GIS. The metadata and job states are crucial for GIS functioning and they must be secured on reserve site almost immediately. The rest persistent data (mainly historical) could be delivered with some delay. Fallen off GIS in short time must be replaced with relevant up-to-date GIS instance. In lightweight grid this is especially important because the GIS is running on non-dedicated server.

4.4 Dynamic deployment of components

Following the line of simplicity, the dynamic deployment of components is very promising. An implementation in which GIS can deploy itself without any human intervention is the ideal case.

4.5 Reconfiguration and adaptability

The ability of platform to self-organise itself is a vital prerequisite for resilience to failures. The platform should give an opportunity for dynamic addition and removal of GIS instances as an appropriate reaction to environmental conditions.
In supporting reconfiguration and adaptability, the platform may utilise rules and knowledge gathered across runs.

4.6 Performance

Considering that GIS will reside on a site which is not dedicated only to that service, it must be lightweight, with minimal impact on the host, efficient and fast.

5 Evaluation of existing information technologies

As already mentioned, the most flexible decision for GIS relies on common infrastructure that merges static information and dynamic monitoring data and provides a uniform schema for both operational and historical information. Grid Monitoring Architecture (GMA), proposed by Global Grid Forum, models grid's information as a set of producers (that provide information), consumers (that request information) and a single registry (integrated repository that mediates the interaction between producers and consumers). Furthermore, the GMA supports rich set of interfaces for data exchange: publish/subscribe; query/response and event notification.
Currently, there are two main approaches to information representation: hierarchical and relational. The hierarchical model models objects as entries in trees. One of the objects is a root of the tree and the rest are related to it. It supports only one type relationship: parent-to-child ("is-a"). The relational model represents object type as relation; the objects are themselves tuples of the relation. It allows peer-to-peer relationships ("part-of") between relations.

5.1 LDAP

The Lightweight Directory Access Protocol (LDAP) is an open network protocol for querying and modifying directory services running over TCP/IP. The LDAP namespace is based on entries which are joined in hierarchical Directory Information Tree. LDAP is designed for information retrieval rather than for updating information, and offer a static view of the data.
The Globus Toolkit provides application developers with a hierarchical GIS infrastructure called Metadata Directory Service (MDS2) using LDAP. It consists of two components: Grid Resource Information Service (GRIS) and Grid Index Information Service (GIIS). The GRIS supplies information about primary resources, while the GIIS aggregates information from multiple GRIS sources and provides a virtual scope for fast searching.

5.1.1 Advantages of using LDAP:

- It is an open standard (i.e. vendor independent).
- It is a widespread protocol.
- The information model is extensible, although radical changes are not possible.
- Directories can be distributed and replicated to provide scalability and reliability.

- High level of data protection and data access: every LDAP server is managed by a domain administrator who defines its security policies.
- Data access is via standardised Internet protocols.

5.1.2 Disadvantages of using LDAP:

- LDAP has very poor update performance and can not efficiently managed huge amount of data.
- LDAP, as well as other hierarchical structures, is fine if all queries are known in advance. In this case the schema can be built to answer them very quickly. However, if you fail to anticipate the questions, getting an answer could be very expensive.
- The LDAP query language is limited. There is no join operation.

5.2 Relational approach

Relational Grid Monitoring Architecture (R-GMA) presents the information resources as a virtual database containing a set of tables with complex relationships among them. Note that R-GMA is not distributed relational DBMS but rather a way to use relational in distributed environment (consistency requirement is not so emphasized in some cases).

Besides producers and consumers, R-GMA provides two other components: schema and registry. A schema contains the name and structure (column names, types and description) of each virtual table in the system. A registry contains a list, for each table, of producers who insert rows (publish data) for that table i.e. it maps producers to the logical database table they produce information for. On registration the producer uses SQL WHERE predicate to define which partition of table data it will populate. The data itself is inserted into logical table through SQL INSERT statement. There is also an archive component (secondary producer) that combines and filters information streams from multiple producers and archive them in a database.

The consumers run SQL SELECT query on the virtual database in order to fetch necessary records. The query is first checked against the registry to identify which producers, for each virtual table in the query, must be contacted. The query is then passed to each relevant producer, to obtain the answer tuples directly.

There is no central repository holding the contents of the tables; it is in this sense virtual database.

5.2.1 Advantages of using relational approach:

- The main advantage is that the infrastructure relies on relational DBMS, which is a proven and effective technology for both fast reading and writing.
- It uses powerful structural query language (SQL).
- It is extendable model (additional information can be added without any problem).
- There are lots of people familiar with relational DBMS products.

5.2.2 Disadvantages of using relational approach:

- Relational DBMS is based on strict ACID principles. In GIS, however, they are not so essentials whereas the strong conditions could lead to performance problems.
- Some of the systems are too complex and too hard for administration.
- When DBMS saves information from multiple administrative domains, the data partitioning for the implementation of a security policy is a more difficult task.

5.3 UDDI

UDDI, the Universal Description Discovery and Integration, is a Web Services registry standard, which provides a data model for describing businesses services. There is also the Universal Business Registry (UBR), which is a global public registry for advertising the available services. In addition UDDI can be set up as a private or community registry. However, it is a technology suitable mostly for static data.
The Globus' Monitoring and Discovery System (MDS4) is a suite of web services to monitor and discover resources and services on Grids. MDS4 includes two Web Services Resource Framework (WSRF)-based services: an Index Service, that gathers data from various sources and a Trigger Service, which can be configured to take action when pre-defined trouble conditions are met. The Index Service is a registry similar to UDDI, but much more flexible. Clients use standard WSRF resource property query and subscription/notification interfaces to retrieve information from an Index. Indexes can register to each other in a hierarchical fashion in order to aggregate data at several levels. An additional Archive Service will provide access to historical data.

5.3.1 Advantages of using UDDI (UDDI-similar):

- It is an open and increasingly supported standard.
- It is flexible and scalable.

5.3.2 Disadvantages of using UDDI (UDDI-similar):

- There are difficulties in handling dynamic information that requires frequent updating.
- There is no explicit data typing for information in the UDDI directory.
- It supports useful but somewhat limited query language.

6 Integration within the STE Institute

One of the main objectives of the CoreGRID Institute on Grid Systems, Tools, and Environments (STE) – work package 7 (WP7) – is to design a generic, lightweight Grid platform, based on extensible component technology.
Following the STE roadmap we present an overview of technically demand for GIS in lightweight Grid environment and examine how existing platforms could be in order to be built generic, portable, reliable and easy maintaining GIS that resides on non-dedicated servers and uses free software.

7 Conclusions

The choice of suitable information model is a key step to the successful deployment and use of the Grid infrastructure. The review of the existing information technologies can help in that aim. The hierarchical data model is well suited to the distribution needs of the computational grid because the rooted tree lends itself well to partitioning into subtrees by administrative domains. Partitioning a RDBMS is not as intuitive; it requires partitioning tables by rows. UDDI has a great potential to become a reliable platform for GIS. There are many considerations that have to be taken into account: data distribution, replication, consistency, etc.
Choosing relevant information technology and data model for lightweight grid is even more complicated because of the requirements for simplicity, good functionality and efficiency.

References

[1] Beth Plale, Peter Dinda Gregor von Laszewski, “Key Concepts and Services of a Grid Information Service”
[2] Rosa M. Badia, Olav Beckmann, Marian Bubak, Denis Caromel, Vladimir Getov, Stavros Isaiadis, Vladimir Lazarov, Maciek Malawski, Sofia Panagiotidi, Jeyarajan Thiyagalingam, “Lightweight Grid Platform: Design Methodology”, October 2005
[3] Xuehai Zhang, Jeffrey Freschl, Jennifer M. Schopf, “A Performance Study of Monitoring and Information Services for Distributed Systems”
[4] Rob Byrom, Brian Coghlan, Andy Cooke, Roney Cordenonsi, Linda Cornwall, Martin Craig, Abdeslem Djaoui, Steve Fisher, Alasdair Gray, Steve Hicks, Stuart Kenny, Jason Leake, Oliver Lyttleton, James Magowan, Robin Middleton, Werner Nutt, David O'Callaghan, Norbert Podhorszki, Paul Taylor, John Walk, Antony Wilson, “Production Services for Information and Monitoring in the Grid”
[5] Brian Coghlan, Abdeslem Djaoui, Steve Fisher, James Magowan, Manfred Oevers, “Time, Information Services and the Grid”, May 2001
[6] Peter Dinda, Beth Plale, “A Unified Relational Approach to Grid Information Services”, February 2001
[7] Giovanni Aloisio, Massimo Cafaro, Italo Epicoco, Sandro Fiore, “Analysis of the Globus Toolkit Grid Information Service”
[8] Rob Allan, Dharmesh Chohan, Xiao Dong Wang, Andy Richards, Mark McKeown, John Colgrave, Matthew Dovey, Mark Baker, Steve Fisher, “Building the e-Science Grid in the UK: Grid Information Services”
[9] Edward Benson, Glenn Wasson, Marty Humphrey, “Evaluation of UDDI as a Provider of Resource Discovery Services for OGSA-based Grids”, April 2006
[10] Petr Holub, Martin Kuba, Ludek Matyska, Miroslav Ruda, “Grid Infrastructure Monitoring as Reliable Information Service”
[11] Andy Cooke, Alasdair Gray, Lisha Ma, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Rob Byrom, Laurence Field, Steve Hicks, Jason Leake, Manish Soni, Antony Wilson, Roney Cordenonsi, Linda Cornwall, Abdeslem Djaoui, Steve Fisher, Norbert Podhorszki, Brian Coghlan, Stuart Kenny, and David O’Callaghan, “R-GMA: An Information Integration System for Grid Monitoring”
[12] Giovanni Aloisio, Euro Blasi, Massimo Cafaro, Italo Epicoco, Sandro Fiore, Maria Mirto, "Dynamic Grid Catalog Information Service"

Multiple Broker Support by Grid Portals*

Extended Abstract

Attila Kertesz[1,3], Zoltan Farkas[1,4], Peter Kacsuk[1,4], Tamas Kiss[2,4]
[1]*MTA SZTAKI Computer and Automation Research Institute*
H-1518 Budapest, P. O. Box 63, Hungary
[2] *Centre for Parallel Computing, University of Westminster*
115 New Cavendish Street, London, W1W 6UW
[3]*CoreGrid Institute on Resource Management and Scheduling*
[4]*CoreGrid Institute on Grid Systems, Tools and Environments*
{attila.kertesz, zfarkas, kacsuk}@sztaki.hu
kisst@wmin.ac.uk

1. Introduction

End-users typically access grid resources through portals that serve as both application developer and executor environments. As grid technology matures the number of production grids dynamically increases. Unfortunately, today's grid portals are typically tightly coupled to one specific grid environment and do not provide multi-grid support. Even if a portal is connected to multiple grids, applications that utilize services from these grids simultaneously are not supported. Meanwhile grids can be realized relatively easily by building a uniform middleware layer, such as Globus [1], on top of the hardware and software resources, the programming concept of such distributed systems is not obvious. Complex problems often require the integration of several existing sequential and parallel programs into a single application in which these codes are executed according to a graph, called workflow. The components of the workflows are jobs, which need to be transferred and executed on the Grid. This task requires special skills, such as how to find out the actual state of the grid, how to reach the resources, etc. To free the users from these efforts, various Resource Brokers [2] have been developed. Though most of these brokers can be used as standalone applications, incorporating them into portals would provide an easier and wider utilization of Grid resources.

There are many grid workflow management systems being developed, such as the Condor DAGMan [1], GridFlow [9] and GWES [10], which are used by different grid portals. Pegasus [8] is a Web-based grid portal. Based on a special configuration file, filled up by the portal administrator with Globus GRAM and GridFTP [1] site addresses, Pegasus is able to map abstract workflows onto physical resources. At the same time – because of the centrally managed resource list and the single certificate the manager applies during workflow execution – Pegasus cannot be considered a multi-grid portal. The GridFlow portal [9] applies a more complex agent-based workflow executor subsystem. The workflow manager of GridFlow handles workflows at two levels: it manages workflows at a global grid level and schedules them at the level of different local grids. Though it has the ability to handle brokers of different grids, it is only coupled to the Globus middleware. In the K-Wf Grid Project [10] the GWES is used as a workflow enactor. They use GWorkflowDL, which is an extension of the GJobDL XML-based language that makes use of the formalism of Petri nets in order to describe the dynamic behavior of distributed Grid jobs. With this

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

approach they are able to model arbitrary algorithms, including parallel branches or loops. It provides methods to initiate and analyze workflows, and to coordinate and optimize the execution of these workflows. They use an own brokering service (Job Handler), which can be utilized on their underlying Globus middlewares.

2. P-GRADE Portal

The P-GRADE Portal [4] is a workflow-oriented multi-grid portal with the main goal to support all stages of grid workflow development and execution processes. It enables the graphical design of workflows created from various types of executable components (sequential, MPI or PVM jobs), executing these workflows in multiple Globus-, and EGEE-based computational grids relying on user credentials, and finally, analyzing the monitored trace-data by its built-in visualization facilities.

Every workflow-oriented portal consists of a workflow GUI and a workflow manager part. A P-GRADE Portal workflow is a directed acyclic graph that connects sequential and parallel programs into an interoperating set of jobs. The nodes of such a graph are batch jobs, while the arc connections define data relations among these jobs. Arcs define the execution order of the jobs and the input/output dependencies that must be resolved by the workflow manager during execution.

The P-GRADE Portal contains a DAGMan-based [7] workflow manager subsystem which is responsible for the scheduling of workflow components in grids. DAGMan degrades workflows into elementary file transfer and job submission tasks and schedules the execution of these tasks. The semantics of the workflow execution means that a node (job) of the workflow can be executed if, and only if all of its input files are available, i.e., all the jobs that produce input files for this job have successfully terminated, and all the other input files are available on the Portal Server and at the pre-defined storage resources. Therefore, the workflow describes both the control-flow and the data-flow of the application. If all the necessary input files are available for a job, then the workflow manager transfers these files - together with the binary executable - to the computational resource where the job is allocated for execution.

3. Resource Management through Brokers

One very important aspect of utilizing multiple Grids within one complex scientific workflow is to interact with multiple and potentially different grid brokers supported by the different Grids. The P-GRADE Portal is interfacing several grid brokers to reach the resources of different grids in an automated way.

As the workflow managers do the actual job submissions, they should be set to utilize brokers. In P-GRADE Portal DAGMan is responsible for workflow execution. Although it itself cannot invoke grid services, it supports customized grid service invocations by its pre/scheduler/post script concept [7]. One pre and one post script can be attached to every node (job) of a DAGMan workflow. DAGMan guarantees, that it first executes the pre script, then the actual content script and finally the post script when it reaches a workflow node. Consequently, the Portal Server automatically generates appropriate pre, content and post scripts for every workflow node when the workflow is saved on the server. These scripts - started by DAGMan according to the graph structure -, invoke the GridFTP and GRAM clients to access files and start up jobs in the connected grids. DAGMan invokes these scripts in the same way in both single- and multi-grid configuration. In general, when a broker is used for job submission, the pre script prepares the broker utilization, the scheduler script invokes the broker, and the post script waits till the execution is finished. The broker provides information about the actual job status and the post script notifies the portal about the status changes.

Currently the portal can utilize GTbroker [6] for Globus 2, 3 and LCG-2 grids, the WMS of the LCG-2 and gLite middlewares [3], and the broker of NorduGrid middleware [5]. The jobs of the workflow that require EGEE services can run on an EGEE type of grid; jobs that require only Globus services can be mapped to resources handled by GTbroker, and finally the NorduGrid Broker can be utilized to reach resources of the NorduGrid ARC [5]. Different Resource Brokers usually require different user job descriptions. In the Workflow Editor of the portal the users can choose a broker for each job of the defined workflow. According to this setting the Editor generates an RSL [1], JDL [3] or an xRSL [5] file from the job requirements, depending on the appropriate broker utilization. The scheduler script of DAGMan invokes the brokers with these descriptions. In case of Globus-based grids the file movements are also handled by GTbroker, so the scheduler script only needs to execute the broker. In case of EGEE WMS and NorduGrid broker, there are special commands for tracking job states and retrieving the output and log files, therefore the scheduler script needs to call these services, too. In case of remote files only the EGEE brokers use a so called ‘close to file’ policy: it means they try to place the job to a resource, which is the nearest to the location of the storage element of its remote input files. Since none of these brokers handle remote file transfers, the execution service of the portal uses a trick: DAGMan submits a wrapper script as the executable, carrying all the job files and descriptions. After this script is started on the selected computing resource, it handles the remote input file transfers and – after the real job execution – the remote output file transfers between the storage elements and the actual computing element. With this solution all kinds of file transfers can be carried out during broker utilization.

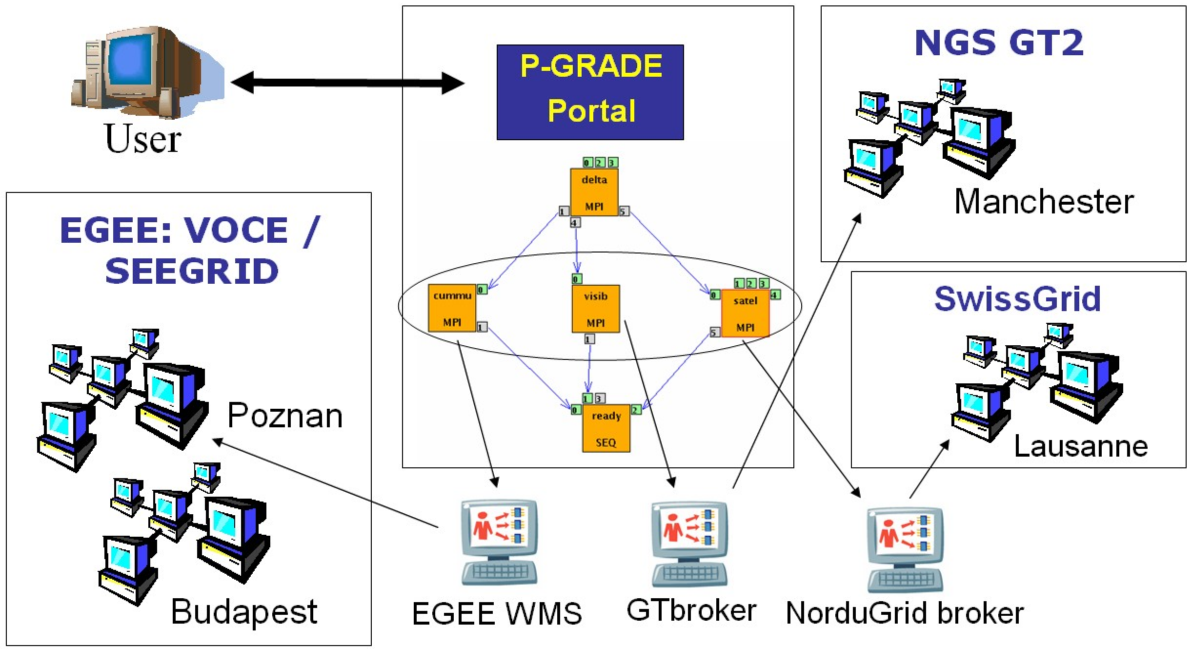


Figure 1. Multiple Broker utilization

In Figure 1., it can be seen, how multiple broker utilization is carried out in the P-GRADE Portal. This way a workflow can be brokered over several grids based on different underlying technologies but still providing optimal utilization of resources. As the portal is able to adopt other brokering services in a similar way, P-GRADE portal users can utilize a growing number of Grids for highly computation intensive applications in the future.

4. Integration within the STE Institute

The P-GRADE Portal is serving the users of the largest production Grids all around the world. The multiple broker utilization of the Portal provides a solution that enables easier and more efficient resource usage. The achieved results thus represent a significant link between Coregrid and Grid end users representing research and industry. Condor DAGMan enactment engine is used by the P-GRADE Portal for workflow management purposes. DAGMan has been extended by our group with several middleware specific scripts to make brokering, job submission and file management possible from the Portal into Globus, LCG, gLite and the NorduGrid ARC middleware based Grids. These scripts act as gateways between the users' applications and standard production level middleware services. Coregrid task 7.3 is working on a high level toolkit, which will establish an abstract layer on top of component based Grid frameworks, primarily on top of the component framework and mediator components under development by Coregrid tasks 7.1 and 7.2. Our current work exactly fits in this process; the P-GRADE Portal as a multi-grid portal uses the underlying brokering services in order to provide an easy-to-use access to the "invisible Grid". However, this integration is reasonable only if we do not loose any of the existing portal features, thus if the toolkit is able to interoperate not only with Coregrid specific, but also with the widely used production Grid services. In this way the collective result of Coregrid WP7 would represent an important link towards projects and consortiums working on component based middleware systems.

5. Conclusions

The P-GRADE Portal gives a Globus-based implementation for workflow management. With exploiting the advanced workflow management features of the P-GRADE portal and the brokering functions of the interconnected Resource Brokers, users can develop and execute multi-grid workflows in a convenient environment. Users have access to more VOs can create such multi-grid workflows that reach resources from even different grids. Furthermore, the execution is carried out in an efficient, brokered way. Since almost every production grid uses Globus and EGEE middlewares today, these grids could all be accessed by the P-GRADE Portal and the workflows created by the portal can produce the expected results.
P-GRADE Portal [18] has been already connected to several European grids (LHC Grid [3], EU GridLab testbed [11], UK OGSA test-bed [12], UK NGS [13], SwissGrid [17]) and serves as a graphical interface for several production grids like SEE-GRID [14], VOCE [15] and HunGrid [16]. As only the P-GRADE portal supports real multi-grid utilization so far, it has been chosen to be the official GIN (Grid Interoperability Now) VO portal for resource testing [19].

6. References

[1] I. Foster C. Kesselman, "The Globus project: A status report", in Proc. of the Heterogeneous Computing Workshop, IEEE Computer Society Press, 1998, pp. 4-18.
[2] A. Kertesz, P. Kacsuk, "A Taxonomy of Grid Resource Brokers", 6th Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS), Innsbruck, Austria, 2006.
[3] LCG-2 User Guide, 4 August, 2005: https://edms.cern.ch/file/454439/2/LCG-2-UserGuide.html; gLite Middleware: http://glite.web.cern.ch/glite/

[4] Cs. Nemeth, G. Dozsa, R. Lovas, P. Kacsuk, “The P-GRADE Grid Portal”, Lecture Notes in Computer Science, Volume 3044, Jan 2004, pp. 10-19.
[5] NorduGrid Middleware: http://www.nordugrid.org/middleware/
[6] A. Kertesz, “Brokering solutions for Grid middlewares”, in Pre-proc. of 1st Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, 2005.
[7] D. Thain, T. Tannenbaum, and M. Livny, “Distributed Computing in Practice: The Condor Experience”, Concurrency and Computation: Practice and Experience, 2005, pp. 323-356.
[8] G. Singh et al, “The Pegasus Portal: Web Based Grid Computing” In Proc. Of 20th Annual ACM Symposium on Applied Computing, Santa Fe, New Mexico, 2005.
[9] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd, “GridFlow: WorkFlow Management for Grid Computing”, In Proc. of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'03), 2003, pp. 198-205.
[10] Falk Neubauer, Andreas Hoheisel and Joachim Geiler, “Workflow-based Grid applications”, Future Generation Computer Systems, Volume 22, Issues 1-2, January 2006, pp. 6-15.
[11] G. Allen et. al., “Enabling Applications on the Grid: A GridLab Overview”, International Journal of High Performance Computing Applications, Issue 17, 2003, pp. 449-466.
[12] UK e-Science OGSA Testbed: http://dsg.port.ac.uk/projects/ogsa-testbed/
[13] UK National Grid Service: http://www.ngs.ac.uk/
[14] Southern Eastern European GRid-enabled eInfrastructure Development (SEE-GRID): http://www.see-grid.org/
[15] Virtual Organisation for Central Europe (VOCE): http://egee.cesnet.cz/en/voce/
[16] The HunGrid Virtual Organisation: http://www.lcg.kfki.hu/?hungrid
[17] The Swiss Grid Initiative: http://www.swiss-grid.org/index.html
[18] P-GRADE Grid Portal: http://lpds.sztaki.hu/pgportal
[19] P-GRADE GIN VO Portal: https://gin-portal.cpc.wmin.ac.uk:8080/gridsphere