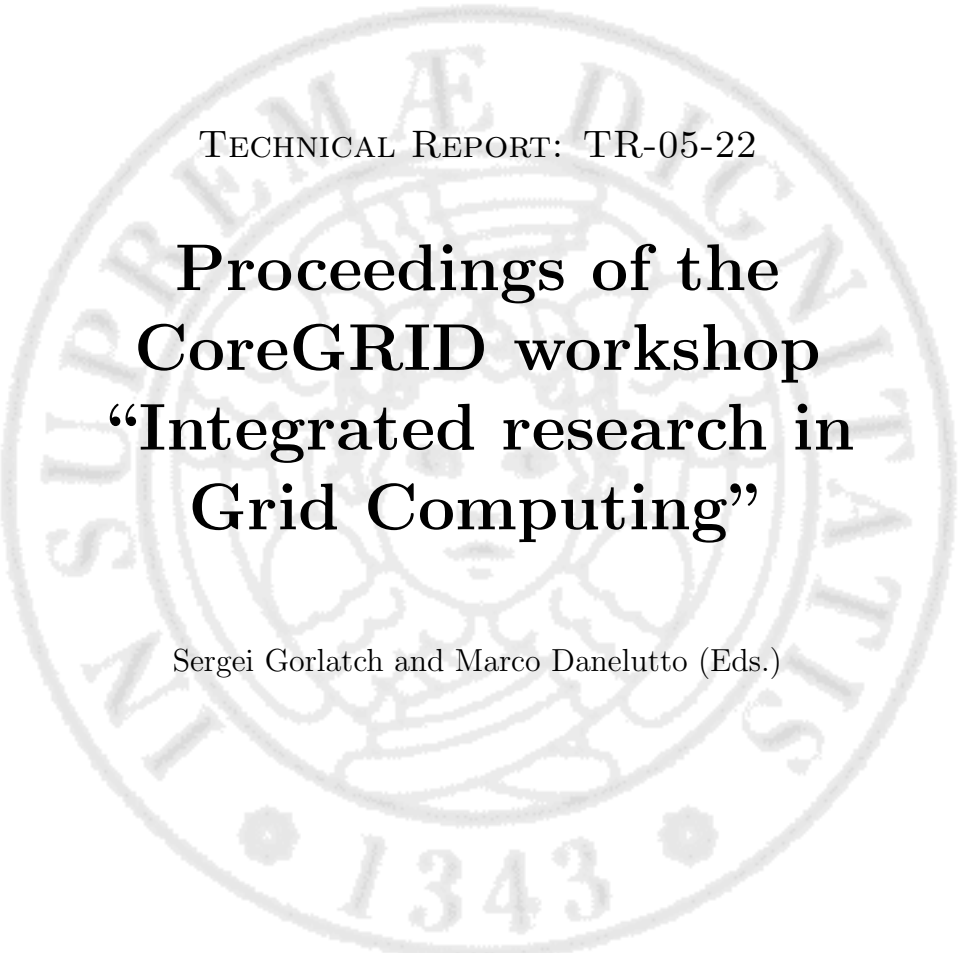
Università di Pisa
Dipartimento di Informatica



Technical Report: TR-05-22

Proceedings of the CoreGRID workshop "Integrated research in Grid Computing"

Sergei Gorlatch and Marco Danelutto (Eds.)

November 28–30, 2005

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Foreword

The CoreGRID Integration Workshop (CGIW'2005) took place on 28-30. November 2005 in Pisa, Italy.

The workshop is organised by the Network of Excellence CoreGRID funded by the European Commission under the sixth Framework Programme IST-2003-2.3.2.8 starting September 1st, 2004. CoreGRID aims at strengthening and advancing scientific and technological excellence in the area of Grid and Peer-to-Peer technologies. To achieve this objective, the network brings together a critical mass of well-established researchers (119 permanent researchers and 165 PhD students) from forty-two institutions who have constructed an ambitious joint programme of activities.

The goal of the workshop is to promote the integration of the CoreGRID network and of the European research community in the area of Grid technologies, in order to overcome the current fragmentation and duplication of efforts in this area.

The list of topics of Grid research covered at the workshop includes but is not limited to:

- knowledge & data management;
- programming models;
- system architecture;
- Grid information, resource and workflow monitoring services;
- resource management and scheduling;
- systems, tools and environments;
- trust and security issues on the Grid.

Priority at the workshop is given to work conducted in collaboration between partners from different research institutions and to promising research proposals that can foster such collaboration in the future.

The workshop is open to the participants of the CoreGRID network and also to the parties interested in cooperating with the network and/or, possibly joining the network in the future.

The Programme Committee who made the selection of papers includes:

Sergei Gorlatch, University of Muenster, Chair
Marco Danelutto, University of Pisa
Domenico Laforenza, ISTI-CNR
Uwe Schwiegelshohn, University of Dortmund
Thierry Priol, INRIA/IRISA
Artur Andrzejak, ZIB
Vladimir Getov, University of Westminster
Ludek Matyska, Masaryk University Brno
Domenico Talia, Universita' della Calabria

Ramin Yahyapour, Universität Dortmund
Norbert Meyer, Poznan Supercomputing and Networking Center
Pierre Guisset, CETIC
Wolfgang Ziegler, Fraunhofer-Institute SCAI
Bruno Le Dantec, ERCIM

The Workshop Organising Committee includes:

Marco Danelutto, University of Pisa
Martin Alt, University of Muenster
Sonia Campa, University of Pisa
Massimo Coppola, ISTI/CNR

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

We gratefully acknowledge the support from the members of the Scientific Advisory Board and Industrial Advisory Board of CoreGRID. Special thanks are due to the authors of all submitted papers, the members of the Programme Committe and the Organising Committee, and to all reviewers, for their contribution to the success of this event. We are grateful to the University of Pisa for hosting the Workshop and publishing its preliminary proceedings.

Pisa, November 2005

Sergei Gorlatch
Marco Danelutto

Contents

Referees

Martin Alt
Artur Andrzejak
Mehmet Ceyran
Marco Danelutto
Jan Dünnweber
Pierre Guisset
Felix Hupfeld
Norbert Meyer
Jens Müller
Thomas Röblitz
Florian Schintke
Thorsten Schütt
Domenico Talia
Ramin Yahyapour
Wolfgang Ziegler

Data integration and query reformulation in service-based Grids

Carmela Comito[1], Anastasios Gounaris[2], Rizos Sakellariou[2], and Domenico Talia[1]

[1] DEIS, University of Calabria, Italy
{ccomito,talia}@deis.unical.it
[2] School of Computer Science, University of Manchester, UK
{gounaris,rizos}@cs.man.ac.uk

Abstract. This paper firstly summarises the work thus far on the XMAP data integration framework and query reformulation algorithm and on middleware with regard to Grid query processing services, namely OGSA-DQP. Secondly, it proposes an architecture for data integration-enabled query processing on the Grid, and finally, it presents a roadmap for its implementation with a view to producing an extended set of e-Services. These services will allow users to submit queries over a single database and receive the results from multiple databases that are semantically correlated with the former one.

1 Introduction

The Grid offers new opportunities and raises new challenges in data management that arise from the large scale, dynamic, autonomous, and distributed nature of data sources. A Grid can include related data resources maintained in different syntaxes, managed by different software systems, and accessible through different protocols and interfaces. Due to this diversity in data resources, one of the most demanding issue in managing data on Grids is reconciliation of data heterogeneity[11]. Therefore, in order to provide facilities for addressing requests over multiple heterogeneous data sources, it is necessary to provide data integration models and mechanisms.

Data integration is the flexible and managed federation, analysis, and processing of data from different distributed sources. In particular, the increase in availability of web-based data sources has led to new challenges in data integration systems for obtaining decentralized, wide-scale sharing of data, preserving semantics. These new needs in data integration systems are also felt in Grid settings. In a Grid, a centralized structure for coordinating all the nodes may not be efficient because it can become a bottleneck and, more importantly, it cannot accommodate the dynamic and distributed nature of Grid resources.

The Grid community is devoting great attention toward the management of structured and semi-structured data such as relational and XML data. Two significant examples of such efforts are the *OGSA Data Access and Integration* (OGSA-DAI)[4] and the *OGSA Distributed Query Processor* (OGSA-DQP)[3] projects. However, till today only few projects (e.g., [9, 7]) actually meet schema-integration requirements necessary for establishing semantic connections among heterogeneous data sources.

For these reasons, we propose the use of the *XMAP* framework [10] for integrating heterogeneous data sources distributed over a Grid. By means of this framework, we

aim at developing a decentralized network of semantically related schemas that enables the formulation of distributed queries over heterogeneous data sources. We designed a method to combine and query XML documents through a decentralized point-to-point mediation process among the different data sources based on schema mappings. We offer a decentralized service-based architecture that exposes this XML integration formalism as an e-Service. The infrastructure proposed exploits the middleware provided by OGSA-DQP and OGSA-DAI, building on top of them schema-integration services.

The remainder of the paper is organized as follows. Section 2 presents a short analysis of data integration systems focusing on specific issues related to Grids. Section 3 presents the XMAP integration framework; the underlying integration model and the XMAP query reformulation algorithm are described. The OGSA-DQP and OGSA-DAI existing query processing services are outlined in Section 4. Section 5 presents a simple example of applying the XMAP algorithm to OGSA-DQP supported relational databases, whereas Section 6 deals with implementation details and roadmap. Finally, Section 7 concludes the paper.

2 Background

The goal of a data integration system is to combine heterogeneous data residing at different sites by providing a unified view of this data. The two main approaches to data integration are federated database management systems (FDBMSs) and traditional mediator/wrapper-based integration systems.

A federated database management system (FDBMS)[19] is a collection of cooperating but autonomous component database systems (DBSs). The DBMS of a component DBS, or component DBMS, can be a centralized or distributed DBMS or another FDBMS. The component DBMSs can differ in different aspects such as data models, query languages, and transaction management capabilities.

Traditional data integration systems[17] are characterized by an architecture based on one or more mediated schemas and a set of sources. The sources contain the real data, while every mediated schema provides a reconciled, integrated, and virtual view of the underlying sources. Moreover, the system includes a set of source descriptions that provide semantic mappings between the relations in the source schemas and the relations in the mediated schemas[18].

Data integration on Grids presents a twofold characterization:

1. data integration is a key issue for exploiting the availability of large, heterogeneous, distributed and highly dynamic data volumes on Grids;
2. integration formalisms can benefit from an OGSA-based Grid infrastructure, since it facilitates dynamic discovery, allocation, access, and use of both data sources and computational resources, as required to support computationally demanding database operations such as query reformulation, compilation and evaluation.

Data integration on Grids has to deal with unpredictable, highly dynamic data volumes provided by unpredictable membership of nodes that happen to be participating at any given time. So, traditional approaches to data integration, such as FDBMS [19] and the use of mediator/wrapper middleware[18], are not suitable in Grid settings.

The federation approach is a rather rigid configuration where resources allocation is static and optimization cannot take advantage of evolving circumstances in the execution environment. The design of mediator/wrapper integration systems must be done

globally and the coordination of mediators has been done by a central administrator which is an obstacle to the exploitation of evolving characteristics of dynamic environments. As a consequence, data sources cannot change often and significantly, otherwise they may violate the mappings to the mediated schema.

The rise in availability of web-based data sources has led to new challenges in data integration systems in order to obtain decentralized, wide-scale sharing of semantically-related data. Recently, several works on data management in peer-to-peer (P2P) systems are moving along this direction[5, 8, 13–15]. All these systems focus on an integration approach not based on a global schema: each peer represents an autonomous information system, and data integration is achieved by establishing mappings among the various peers.

To the best of our knowledge, there are only few works designed to provide schema-integration in Grids. The most notable ones are *Hyper*[9] and *GDMS*[7]. Both systems are based on the same approach that we have used ourselves: building data integration services by extending the reference implementation of OGSA-DAI. The *Grid Data Mediation Service* (GDMS) uses a wrapper/mediator approach based on a global schema. GDMS presents heterogeneous, distributed data sources as one logical virtual data source in the form of an OGSA-DAI service. This work is essentially different from ours as it uses a global schema. For its part, *Hyper* is a framework that integrates relational data in P2P systems built on Grid infrastructures. As in other P2P integration systems, the integration is achieved without using any hierarchical structure for establishing mappings among the autonomous peers. In that framework, the authors use a simple relational language for expressing both the schemas and the mappings. By comparison, our integration model follows as Hyper an approach not based on a hierarchical structure, however differently form Hyper it focuses on XML data sources and is based on schema-mappings that associate paths in different schemas.

3 XMAP: A Decentralized XML Data Integration Framework

The primary design goal the XMAP framework is to develop a decentralized network of semantically related schemas that enables the formulation of queries over heterogeneous, distributed data sources. The environment is modeled as a system composed of a number of Grid nodes, where each node can hold one or more XML databases. These nodes are connected to each other through declarative mappings rules.

The XMAP integration [10] model is based on schema mappings to translate queries between different schemas. The goal of a schema mapping is to capture structural as well as terminological correspondences between schemas. Thus, in [10], we propose a decentralized approach inspired from [14] where the mapping rules are established directly among source schemas without relying on a central mediator or a hierarchy of mediators. The specification of mappings is thus flexible and scalable: each source schema is directly connected to only a small number of other schemas. However, it remains reachable from all other schemas that belong to its transitive closure. In other words, the system supports two different kinds of mapping to connect schemas semantically: point-to-point mappings and transitive mappings. In transitive mappings, data sources are related through one or more “mediator schemas”.

We address structural heterogeneity among XML data sources by associating paths in different schemas. Mappings are specified as path expressions that relate a specific

element or attribute (together with its path) in the source schema to related elements or attributes in the destination schema.. The mapping rules are specified in XML documents called XMAP documents. Each source schema in the framework is associated to an XMAP document containing all the mapping rules related to it.

The key issue of the XMAP framework is the XPath reformulation algorithm: when a query is posed over the schema of a node, the system will utilize data from any node that is transitively connected by semantic mappings, by chaining mappings, and reformulate the given query expanding and translating it into appropriate queries over semantically related nodes. Every time the reformulation reaches a node that stores no redundant data, the appropriate query is posed on that node, and additional answers may be found. As a first step, we consider only a subset of the full XPath language.

4 Introduction to Grid query processing services

OGSA-DQP is an open source service-based Distributed Query Processor; as such, it supports the evaluation of queries over collections of potentially remote data access and analysis services. OGSA-DQP uses Grid Data Services (GDSs) provided by OGSA-DAI to hide data source heterogeneities and ensure consistent access to data and metadata. The current version of OGSA-DQP, OGSA-DQP 2.0, uses Globus Toolkit 3.2 for grid service creation and management. Thus OGSA-DQP builds upon an OGSA-DAI distribution that is based on the OGSI infrastructure. In addition, both GT3.2 and OGSA-DAI require a web service container (e.g. Axis) and a web server (such as Apache Tomcat) below them. A forthcoming release of OGSA-DQP, due in fall of 2005, will support the WS-I and WSRF platforms as well.

OGSA-DQP provides two additional types of services, Grid Distributed Query Services (GDQSs) and Grid Query Evaluation Services (GQESs). The former are visible to end users through a GUI client, accept queries from them, construct and optimise the corresponding query plans and coordinate the query execution. GQESs implement the query engine, interact with other services (such as GDSs, ordinary Web Services and other instances of GQESs), and are responsible for the execution of the query plans created by GDQSs.

5 Integrating the XMAP algorithm in service-based Grids: A walk-through example

The XMAP algorithm can be used for data integration-enable query processing in OGSA-DQP. This example aims to show how the XMAP algorithm can be applied on top of the OGSA-DAI and OGSA-DQP services. In the example, we will assume that the underlying databases, of which the XML representation of the schema is processed by the XMAP algorithm, are, in fact, relational databases, like those supported by the current version of OGSA-DQP.

We assume that there are two sites, each holding a separate, autonomous database that contains information about artists and their works. Figure 1 presents two self-explanatory views: one hierarchical (for native XML databases), and one tabular (for object-relational DBMSs).

In OGSA-DQP, the table schemas are retrieved and exposed in the form of XML documents, as shown in Figure 2.

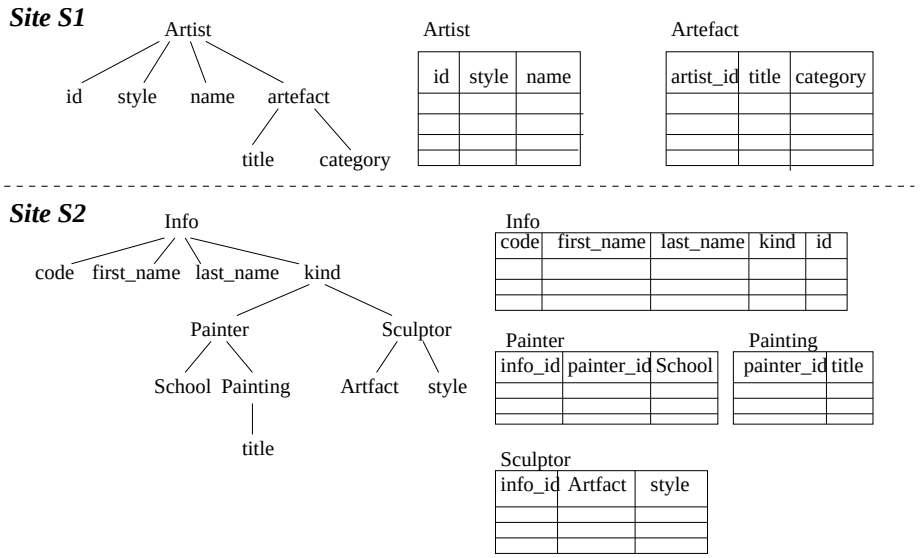


Fig. 1. The example schemas.

The XMAP mappings need to capture the semantic relationships between the data fields in different databases, including the primary and foreign keys. This can be done in two ways, which are illustrated in Figures 3 and 4, respectively. Both the ways seem to be feasible. However, the second one is slightly more comprehensible, and thus more desirable.

The actual query reformulation occurs exactly as described in [10]. Initially, the users submit XPath queries that refer to a single physical database. E.g., `/S1/Artist [style="Cubism''] /name` extracts the names of the artists whose style is Cubism and their data is stored in the *S1* database. Similarly, `/S1/Artefact/title` returns the titles of the artifacts in the same database. When the XMAP algorithm is applied for the second query, two more XPath expressions will be created that refer to the *S2* database: `/S2/Painting/Title` and `/S2/Sculptor/Artefact`. At the back-end, the following queries will be submitted to the underlying databases (in SQL-like format):

```
select title from Artefact;,
select title from Painting;, and
select Artefact from Sculptor;
```

Note that the mapping of simple XPath expressions to SQL/OQL is feasible [16].

6 Implementation Roadmap

In this section we will describe in brief the system design that we envisage, along with the service interactions involved, the implementation strategies and some directions for future research and extensions.

6.1 Service Interactions and System Design

The XMAP query reformulation algorithm is deployed as a stand-alone service, called *Grid Data Integration service (GDI)*. Figure 5 provides an overview of the service interactions involved in the incorporation of data integration functionality in distributed

```
<databaseSchema dbname="S1">
    <table name="Artist">
        <column name="id" />
        <column name="style" />
        <column name="name" />
        <primaryKey>
            <columnName>id</columnName>
        </primaryKey>
    </table>
    <table name="Artefact">
        <column name="artist_id" />
        <column name="title" />
        <column name="category" />
    </table>
</databaseSchema>

<databaseSchema dbname="S2">
    <table name="Info">
        <column name="id" />
        <column name="code" />
        <column name="first_name" />
        <column name="last_name" />
        <column name="kind" />
        <primaryKey>
            <columnName>id</columnName>
        </primaryKey>
    </table>
    <table name="Painter">
        <column name="painter_id" />
        <column name="info_id" />
        <column name="school" />
        <primaryKey>
            <columnName>painter_id</columnName>
        </primaryKey>
    </table>
    <table name="Painting">
        <column name="painter_id" />
        <column name="title" />
        <primaryKey>
            <columnName>title</columnName>
        </primaryKey>
    </table>
    <table name="Sculptor">
        <column name="info_id" />
        <column name="artefact" />
        <column name="style" />
    </table>
</databaseSchema>
```

Fig. 2. The XML representation of the schemas of the example databases.

```
i)   databaseSchema[@dbname=S1]/table[@name=Artist]/column[@name=style] ->
     databaseSchema[@dbname=S2]/table[@name=Painter]/column[@name=school],
     databaseSchema[@dbname=S2]/table[@name=Sculptor]/column[@name=style]
ii)  databaseSchema[@dbname=S1]/table[@name=Artefact]/column[@name=title] ->
     databaseSchema[@dbname=S2]/table[@name=Painting]/column[@name=title],
     databaseSchema[@dbname=S2]/table[@name=Sculptor]/column[@name=artefact]

iii) databaseSchema[@dbname=S1]/table[@name=Artist/column[@name=id ->
     databaseSchema[@dbname=S2]/table[@name=Info/column[@name=id]

iv)  databaseSchema[@dbname=S1]/table[@name=Artefact]/column[@name=artist_id] ->
     databaseSchema[@dbname=S2]/table[@name=Painter]/column[@name=info_id],
     databaseSchema[@dbname=S2]/table[@name=Sculptor]/column[@name=info_id]
```

Fig. 3. The XMAP mappings.

```
i) S1/Artist/style -> S2/Painter/school, S2/Sculptor/style

ii)S1/Artefact/title -> S2/Painting/title, S2/Sculptor/artefact

iii) S1/Artist/id -> S2/Info/id

iv) S1/Artefact/artist_id->S2/Painter/info_id,S2/Sculptor/info_id
```

Fig. 4. A simpler form of the XMAP mappings.

query processing on the Grid. It focuses on the interactions that concern the *GDI*, and thus it hides all the complexities that relate to (distributed) query submission and execution. As such, it complements the service interactions between the OGSA-DAI and DQP services, which are described in detail in [2].

The following architectural assumptions are made. The *GDI* is deployed at each site participating in a dynamic database federation and has a mechanism to load local mapping information. Following the Globus Toolkit 3 [1] terminology, it implements additional *port-types* (see sect 3.2), among which the *Query Reformulation Algorithm(QRA)* port-type, which accepts XPath expressions, applies the XMAP algorithm to them, and returns the results. A database can join the system as in OGSA-DQP: registering itself in a registry and informing the *GDQS*. The only difference is that, given the assumptions above, it should be associated with both a *GQES* and a *GDI*.

Also, there is one *GQES* per site to evaluate (sub)queries, and at least one *GDQS*. As in classical OGSA-DQP scenarios, the *GDQS* contains a view of the schemas of the participating data resources, and a list of the computational resources that are available. The users interact only with this service from a client application that need not be exposed as a service.

The interactions are as follows (see also Figure 5):

1. The client contacts the *GDQS* and requests a view of the schema for each database he/she is interested in.
2. Based on the retrieved schema, he/she composes an XPath query, which is sent to the *GDQS*.
3. The *GDQS* transforms, parses, optimises, schedules and compiles a query execution plan [20]. This process entails the identification of the relevant sites, and consequently their local *GQES* and *GDI*. The resulting query execution plan is sent to the corresponding *GQES*, which returns the results asynchronously, after contacting the local database via a *GDS*.
4. The initial XPath expression is sent to the *GDI* that is co-located with the *GQES* of the previous step to perform the XMAP algorithm.
5. As long as the call to the *GDI* returns at least one XPath expression that has not been considered yet in the same session, the following steps are executed in an iterative manner.
   (a) The results of the call to the *GDI* are collected by the *GDQS*. They contain a set of XPath expressions. The *GDQS* filters out the ones that have already been processed in the current session.
   (b) Each remaining XPath expression is processed as in Step 3 to collect results from other databases than the one initially considered by the user.

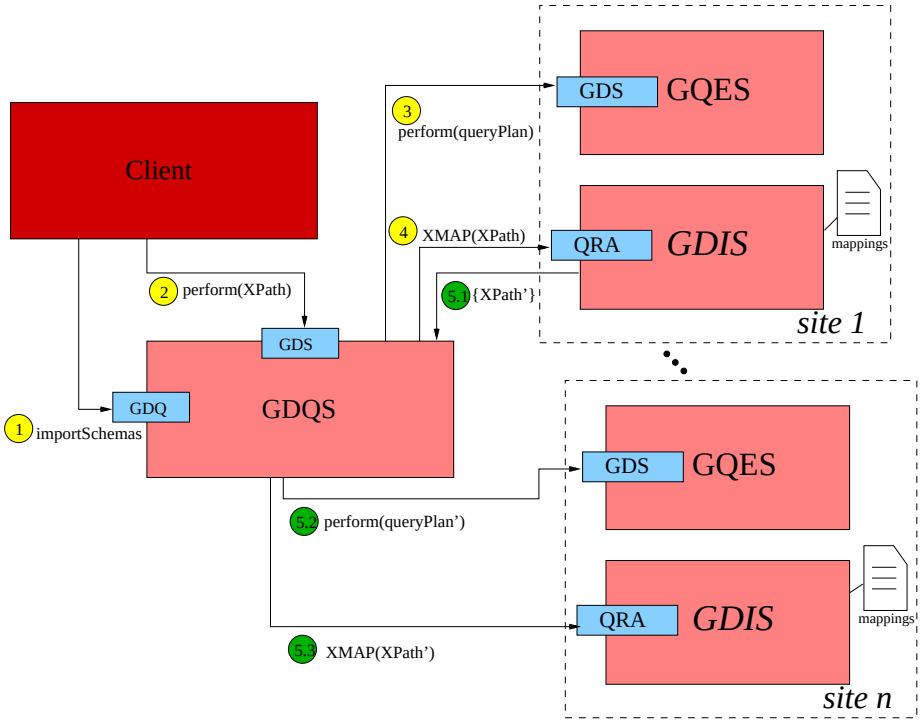


Fig. 5. Data integration-enabled query processing on the Grid: service interactions.

(c) The same XPath expression is processed as in Step 4 to find additional correlated queries.

6.2 A Summary of the Extensions Envisaged to the Current Querying Services

The afore-mentioned architecture, apart from the development of the new *GDI* service, implies some extensions to the current services and clients that are available from OGSA-DAI and OGSA-DQP. These extensions are, in our view, reasonable and feasible, and thus make the overall proposal of practical interest. They are summarised below:

- The client should expose the schemas per database rather than as a unified view.
- *GDQS* should be capable of accepting XPath queries, and of transforming these XPath queries to OQL before parsing, compiling, optimising and scheduling them. Such a transformation falls in an active research area (e.g., [12, 6]), and will be realised as an additional component within the query compiler.
- *GDQS* should implement an additional XMAP-related activity that, given an XPath expression, finds the corresponding *GDI*, and calls the XMAP on it. This returns a set of corresponding XPaths.
- The client should be capable of aggregating results stemming from multiple queries.
- *GDQS* should be capable of accepting requests that contain more than one (XPath) statement.
- Also, *GDI* should be capable of processing requests that clean, update and install mapping documents.

6.3 Looking Ahead

The proposed architecture will provide added value to the existing querying services, and increase the scope of the applications that may use them. It will result in a middleware infrastructure that can be enhanced with more functionality. With a view to incorporating more features, the following stages of extensions have been identified:

Stage A: XPath is a simple language, and, as such, it cannot cover many of the common user requests. Allowing more complex user queries to be submitted, and using the same XMAP algorithm that relies on paths, is a challenging problem. To this end, we are planning to investigate more extensive use of the knowledge about key/foreign-key relationships to reformulate more expressive queries (such as XQuery, SQL and OQL) correctly.

Stage B: OGSA-DQP naturally provides the capability to submit queries over distributed sources in a manner that is transparent to the user. In order to use this functionality in the future, some (non-extensive) changes in the validity criteria of reformulated queries in the XMAP algorithm will be required.

Stage C: A more challenging problem is to allow initial queries to be distributed. This raises a new set of issues, which include which site should hold the mappings, whether any more metadata at the GDQS-level is required, and how non-duplicate results can be guaranteed.

Stage D: Finally, we plan to explore alternative architectures, and especially architectures in which the *GDIs* are not co-located with *GQESs*, and can be shared between multiple sites. Also, we can allow local *GDIs* to contact remote ones directly, in a more peer-to-peer-like fashion.

7 Summary

The contribution of this work is the proposal of an architecture and an approach that integrates a data integration methodology with existing e-Services for querying distributed databases with a view to providing an enhanced, data integration-enabled service middleware. The data integration is based upon the XMAP framework that takes into account the semantic and syntactic heterogeneity between different data resources, and provides a recursive query reformulation algorithm. The Grid services used as a basis are the outcome of the OGSA-DAI/DQP projects, which have paved the way towards uniform access and combination of distributed databases. In summary, in this paper (i) we provide an overview of XMAP and existing querying services, (ii) we show how they can be used together through an example, (iii) we provide a service-oriented architecture to this end, (iv) we discuss implementation issues and (v) we provide insights into how the proposed architecture can be further extended.

References

1. The Globus toolkit, http://www.globus.org.
2. M. Nedim Alpdemir, A. Mukherjee, Norman W. Paton, Paul Watson, Alvaro A. A. Fernandes, Anastasios Gounaris, and Jim Smith. Service-based distributed querying on the grid. In Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors, *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, pages 467–482. Springer, 2003.

3. M. Nedim Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Norman W. Paton, Paul Watson, Alvaro A. A. Fernandes, and Desmond J. Fitzgerald. OGSA-DQP: A service for distributed querying on the grid. In *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology*, pages 858–861, March 2004.
4. Mario Antonioletti and et al. OGSA-DAI: Two years on. In *Global Grid Forum 10 — Data Area Workshop*, March 2004.
5. Philip A. Bernstein, Fausto Giunchiglia, Anastasios Kementsietsidis, John Mylopoulos, Luciano Serafini, and Ilya Zaihrayeu. Data management for peer-to-peer computing : A vision. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB 2002)*, pages 89–94, June 2002.
6. Kevin S. Beyer, Roberta Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Bob Lyle, Fatma Ozcan, Hamid Pirahesh, Norman Seemann, Tuong C. Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System rx: One part relational, one part xml. In *SIGMOD Conference 2005*, pages 347–358, 2005.
7. P. Brezany, A. Woehrer, and A. M. Tjoa. Novel mediator architectures for grid information systems. *Journal for Future Generation Computer Systems - Grid Computing: Theory, Methods and Applications.*, 21(1):107–114, 2005.
8. Diego Calvanese, Elio Damaggio, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Semantic data integration in P2P systems. In *Proceedings of the First International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P)*, pages 77–90, September 2003.
9. Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Riccardo Rosati, and Guido Vetere. Hyper: A framework for peer-to-peer data integration on grids. In *Proc. of the Int. Conference on Semantics of a Networked World: Semantics for Grid Databases (ICSNW 2004)*, volume 3226 of *Lecture Notes in Computer Science*, pages 144–157, 2004.
10. C. Comito and D. Talia. Xml data integration in ogsa grids. In *1st Int. Workshop on Data Management in Grids (to appear)*, 2005.
11. Karl Czajkowski and et al. The WS-resource framework version 1.0. The Globus Alliance, Draft, March 2004. `http://www.globus.org/wsrf/specs/ws-wsrf.pdf`.
12. Wenfei Fan, Jeffrey Xu Yu, Hongjun Lu, and Jianhua Lu. Query translation from xpath to sql in the presence of recursive dtds. In *VLDB Conference 2005*, 2005.
13. Enrico Franconi, Gabriel M. Kuper, Andrei Lopatenko, and Luciano Serafini. A robust logical and computational characterisation of peer-to-peer database systems. In *Proceedings of the First International Workshop on Databases, Information Systems, and Peer-to-Peer Computing (DBISP2P)*, pages 64–76, September 2003.
14. Alon Y. Halevy, Dan Suciu, Igor Tatarinov, and Zachary G. Ives. Schema mediation in peer data management systems. In *Proceedings of the 19th International Conference on Data Engineering*, pages 505–516, March 2003.
15. Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 325–336, June 2003.
16. George Lapis. Xml and relational storage - are they mutually exclusive? available at http://www.idealliance.org/proceedings/xtech05/papers/02-05-01/ (accessed in july 2005).
17. Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 233–246, June 2002.
18. Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB'96)*, pages 251–262, September 1996.
19. Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, 1990.
20. Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. In Manish Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, pages 279–290. Springer, 2002.

Design of Knowledge Discovery Services Using the WS-Resource Framework

Antonio Congiusta, Domenico Talia, and Paolo Trunfio

DEIS
University of Calabria
Via P. Bucci 41C, 87036 Rende (CS), Italy
{acongiusta, talia, trunfio}@deis.unical.it

Abstract. Knowledge discovery in large data sets involves processes and activities that are computational intensive, collaborative, and distributed in nature. The Grid is a profitable infrastructure that can be effectively exploited for handling distributed data mining and knowledge discovery. To achieve this goal, advanced software tools and services are needed to support both the development of KDD applications. The Knowledge Grid is a high-level framework providing Grid-based knowledge discovery tools and services. Such services allow users to create and manage complex knowledge discovery applications that integrate data sources and data mining tools provided as distributed services on a Grid. All of these services are currently being re-implemented as WSRF-compliant Grid Services. This paper highlights design aspects and implementation choices involved in such a process.

1 Introduction

The advent of the Grid has introduced substantial changes in the way data and computations are conceived and developed within industrial and scientific applications. Size limits, administrative boundaries, and data heterogeneity are no longer intractable problems nowadays. As a consequence, even more huge amounts of data are being produced, stored, and moved within Grid systems as a result of data acquisitions from remote instruments, or scientific experiments, simulations, and so forth. Handling and mining large volumes of semi-structured and unstructured data is still the most critical issue currently affecting scientists and companies attempting to make an intelligent and profitable use of their data. One of the present challenges of the Grid is thus making the production and ownership of such data competitive and useful by allowing effective and efficient extraction of valuable knowledge from it. To this end, knowledge discovery and data mining services are needed to help researchers and professionals to analyze the very large amount of data that today is stored in digital formats in file systems, data warehouses and databases distributed over corporate or worldwide Grids. The Knowledge Grid [1] is a framework for implementing knowledge discovery tasks in a wide range of high-performance distributed applications. The Knowledge Grid offers to users high-level abstractions and a set of services by

which is possible to integrate Grid resources to support all the phases of the knowledge discovery process, as well as basic, related tasks like data management, data mining, and knowledge representation. Therefore, it allows end-users to concentrate on the knowledge discovery process they must develop, without worrying about the Grid infrastructure and its low-level details. The framework supports data mining on the Grid by providing mechanisms and higher level services for

- searching resources,
- representing, creating, and managing knowledge discovery processes, and
- composing existing data services and data mining services as structured, compound services,

so as to allow users to plan, store, document, verify, share and (re-)execute their applications, as well as manage their output results. Previous research activities on the Knowledge Grid have been focused on the assessment of the design of the framework and the evaluation of the development process of Grid-based KDD applications, including their performance [5, 6], based on a prototype of the system developed within Grid environment not based on Grid Services.

This paper details design aspects and implementation choices related to the deployment of the Knowledge Grid system using the WS-Resource Framework (WSRF) [3]. WSRF has been chosen because it is the emerging service-based paradigm for the Grid and has shown to well adapt to the implementation of knowledge discovery services (see Section 4).

Section 2 describes the features of the Knowledge Grid; Section 3 discusses the general structure of a K-Grid service along with its invocation mechanisms and WS-Resource management, moreover in depth details about the implementation of the services performing data and tools access are provided. Section 4 reports the performance evaluation of a basic K-Grid service and concludes the paper.

2 The Knowledge Grid Framework

The Knowledge Grid uses basic Grid mechanisms to build specific knowledge discovery services. These services can be implemented in different ways using the available Grid environments such as Globus, UNICORE, etc. This layered approach benefits from "standard" Grid services that are more and more utilized and offers an open, distributed knowledge discovery architecture that can be configured on top of any Grid middleware in a simple way.

The Knowledge Grid has a layered architecture comprising two categories of services. The *High-level K-Grid layer* includes services used to compose, validate, and execute a distributed knowledge discovery computation. The main services of the High-level K-Grid layer are:

- The *Data Access Service* (*DAS*) is responsible for the publication and searching of data to be mined (data sources), and the searching of discovered models (mining results).

– The *Tools and Algorithms Access Service* (*TAAS*) is responsible for the publication and searching of extraction tools, data mining tools, and visualization tools.

– The *Execution Plan Management Service* (*EPMS*). An *execution plan* is represented by a graph describing interactions and data flows between data sources, extraction tools, data mining tools, and visualization tools. In particular, it specifies through a DAG-like notation the application structure in terms of data transfers and data mining tasks. The Execution Plan Management Service allows for defining the structure of an application by building the corresponding execution graph and adding a set of constraints about resources. The execution plan generated by this service is referred to as *abstract execution plan*, because it may include both well identified resources and *abstract resources*, i.e., resources that are defined through constraints about their features, but are not known a priori.

– The *Results Presentation Service* (*RPS*) offers facilities for presenting and visualizing the extracted knowledge models (e.g., association rules, clustering models, classifications).

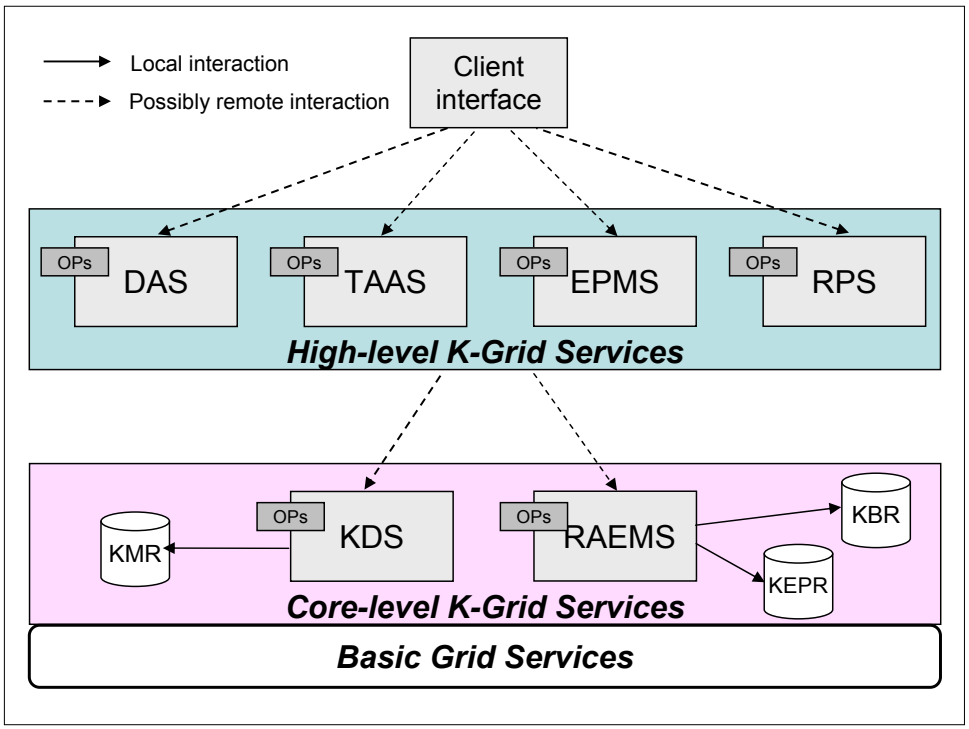


Fig. 1. Interactions between a client and the Knowledge Grid environment.

The *Core K-Grid layer* offers basic services for the management of metadata describing features of hosts, data sources, data mining tools, and visualization tools. This layer coordinates the application execution by attempting to fulfill the application requirements with respect to available Grid resources. The Core K-Grid layer comprises two main services:

- The *Knowledge Directory Service* (*KDS*) is responsible for handling metadata describing Knowledge Grid resources. Such resources include hosts, data repositories, tools and algorithms used to extract, analyze, and manipulate data, distributed knowledge-discovery execution plans, and knowledge models obtained as results of the mining process. The metadata information is represented by XML documents stored in a *Knowledge Metadata Repository* (*KMR*).
- The *Resource Allocation and Execution Management Service* (*RAEMS*) is used to find a suitable mapping between an abstract execution plan and available resources, with the goal of satisfying the constraints (CPU, storage, memory, database, network bandwidth) imposed by the execution plan. The output of this process is an *instantiated execution plan*, which defines the resource requests for each data mining process. Generated execution plans are stored in the *Knowledge Execution Plan Repository* (*KEPR*). After the execution plan activation, this service manages the application execution and the storing of results in the *Knowledge Base Repository* (*KBR*).

3 Knowledge Discovery WSRF-services

The research and industry communities, under the guidance of the *Global Grid Forum* (*GGF*) [2], defined a new standard service-paradigm, the *WS-Resource Framework* (*WSRF*), as an evolution of early OGSA implementations [3]. WSRF codifies the relationships between Web Services and stateful resources in terms of the *implied resource pattern*, which is a set of conventions on Web Services technologies, in particular XML, WSDL, and *WS-Addressing* [4]. A stateful resource that participates in the implied resource pattern is termed as *WS-Resource*. The framework describes the WS-Resource definition and its association with a Web Service interface, and how to make accessible the properties of a WS-Resource through that Web Service interface.

In the WSRF-based implementation of the Knowledge Grid each service is exposed as a Web Service that exports one or more operations (OPs), by using the WSRF conventions and mechanisms. The operations exported by High-level K-Grid services are designed to be invoked by user-level applications only, whereas the operations provided by Core K-Grid services are thought to be invoked by High-level as well as Core K-Grid services.

As shown in Figure 1, users can access the Knowledge Grid functionalities by using a client interface located on their machine. The client interface can be an integrated visual environment that allows for performing basic tasks (e.g., searching of data and software, data transfers, simple job executions), as well as for composing distributed data mining applications described by arbitrarily complex execution plans. The client interface performs its tasks by invoking the appropriate operations provided by the different High-level K-Grid services. Those services may be in general executed on a different Grid node; therefore the interactions between the client interface and High-level K-Grid services are possibly remote.

3.1 K-Grid-service structure

Figure 2 describes the general invocation mechanisms between clients and K-Grid services. Each K-Grid service exports three mandatory operations – `createResource`, `subscribe`, and `destroy` – and one or more service-specific operations. The `createResource` operation is used to create a WS-Resource, which is then used to maintain the state (e.g., results) of the computations performed by the service-specific operations. The `subscribe` operation is used for subscribing to notifications about computation results. The `destroy` operation removes a WS-Resource. Figure 2 shows a generic K-Grid service exporting the mandatory operations and two service-specific operations, `operationX` and `operationY`. A client interacting with the K-Grid service is also shown. Note that, in such a context, a client can be either a client interface or another K-Grid service. The implementation of a K-Grid service follows the WS-Resource fac-

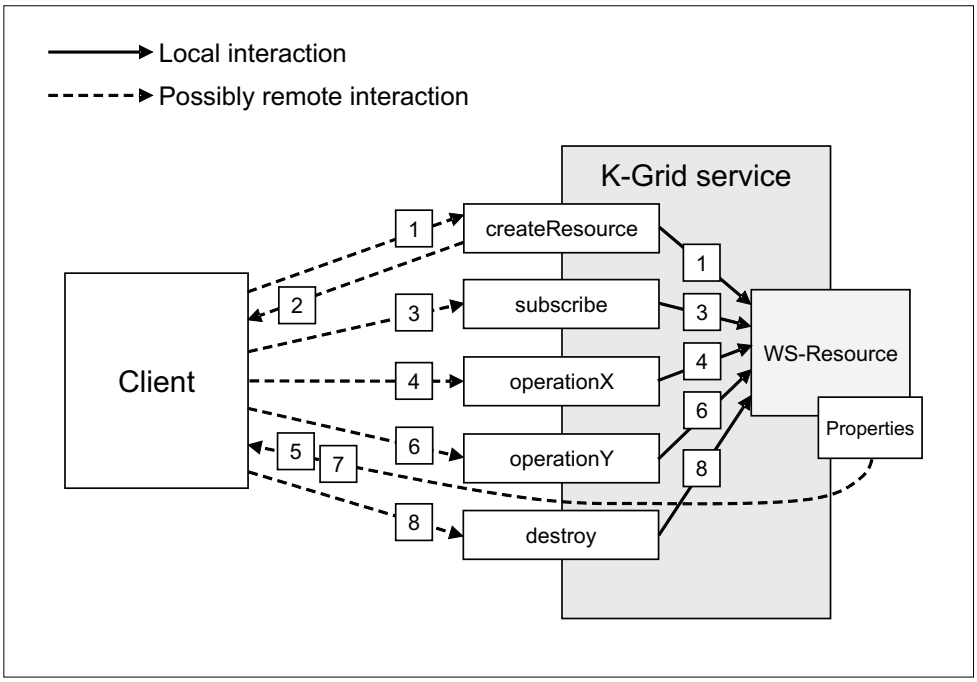


Fig. 2. General K-Grid service invocation mechanism.

tory pattern. In this pattern in order to create a resource, the client contacts the factory service, who will take care of creating and initializing a new resource (see Figure 3). The resource is also assigned a unique key (endpoint reference, EPR for short), the factory service will return an EPR of the WS-Resource composed by the instance service and the recently created resource. From this moment on, the instance service is able to operate on the resource for performing its operations. Table 1 shows the services, and their main associated operations, of the Knowledge Grid. Each of them is a *K-Grid Service* as previously defined. For each operation a description about its activities and interactions with other services is given. The following subsection gives a more detailed discussion of

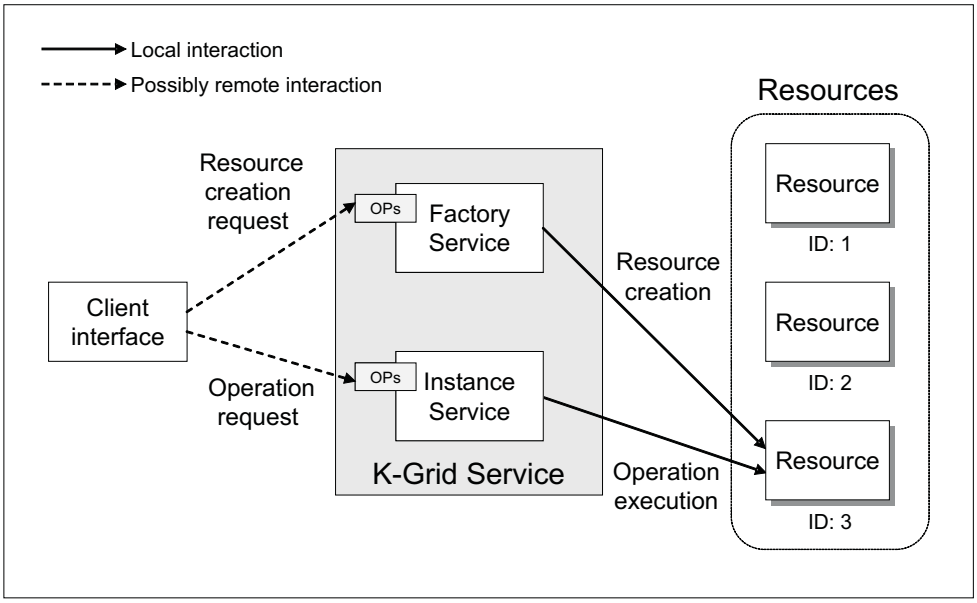


Fig. 3. K-Grid service design

services and operations implementation for a group of services concerning data and tools access tasks.

3.2 Data and Tools access

DAS and TAAS services are concerned with the publishing and searching of data sets and tools to be used in a KDD application. They possesses the same basic structure and performs their main tasks by interacting with a local instance of the KDS that in turn may invoke one or more other remote KDS instances.

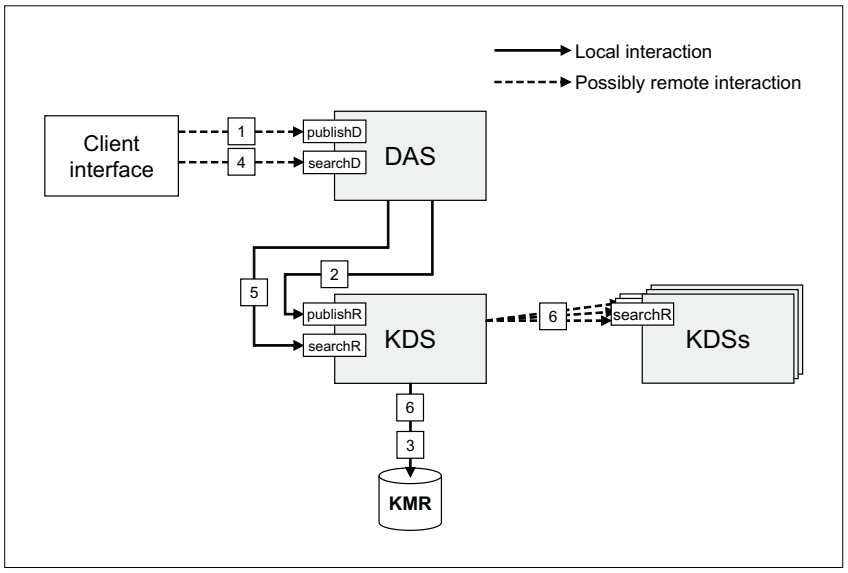


Fig. 4. DAS interactions.

Table 1. Description of main K-Grid service operations.

Service	Operation	Description
DAS	`publishData`	This operation is invoked by a client for publishing a newly available dataset. The publishing requires a set of information that will be stored as metadata in the local KMR.
	`searchData`	The search for available data to be used in a KDD computation is accomplished during the application design by invoking this operation. The searching is performed on the basis of appropriate parameters.
TAAS	`publishTools`	This operation is used to publish metadata about a data mining tool in the local KMR. As a result of the publishing, a new DM service is made available for utilization in KDD computations.
	`searchTools`	It is similar to the searchData operation except that it is targeted to data mining tools.
EPMS	`submitKApplication`	This operation receives a conceptual model of the application to be executed. The EPMS generates a corresponding abstract execution plan and submits it to the RAEMS for its execution.
RPS	`getResults`	Retrieves results of a performed KDD computation and presents them to the user.
KDS	`publishResource`	This is the basic, core-level operation for publishing data or tools. It is thus invoked by the DAS or TAAS services for performing their own specific operations.
	`searchResource`	The core-level operation for searching data or tools.
RAEMS	`manageKExecution`	This operation receives an abstract execution plan of the application. The RAEMS generates an instantiated execution plan and manages its execution.

Figure 4 describes the interactions that occur when the DAS service is invoked; similar interactions apply also to TAAS invocations, therefore we avoid to replicate the figure and the service invocation description.

The `publishData` operation is invoked to publish information about a data set (step 1). The DAS passes the corresponding metadata to the local KDS, by invoking the `publishResource` operation (step 2). The KDS, in turn, stores that metadata into the local KMR (step 3).

The `searchData` operation is invoked by a client interface that needs to locate a data set on the basis of a given set of criteria (step 4). The DAS submits its request to the local KDS, by invoking the corresponding `searchResource` operation (step 5). As mentioned before, the KDS performs the searching both accessing the local KMR, and querying remote KDSs (step 6). This is a general rule enforced within all the interactions between a high-level service and the KDS when a searching is requested. The local KDS is thus responsible for dispatching the query to remote KDSs and for generating the final answer.

Figure 5 shows an excerpt of the WSDL declarations associated with the DAS operations. The `searchString` and `searchResponse` types definitions re-

```
<types>
     ...
     <xsd:element name="ArrayOfString">
     <xsd:complexType >
         <xsd:sequence>
             <xsd:element name="x" type="xsd:string"
                     minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
     </xsd:complexType>
     </xsd:element>

     <!-- REQUESTS AND RESPONSES -->

     <xsd:element name="searchString" type="xsd:string"/>
     <xsd:element name="searchResponse" ref="tns:ArrayOfString"/>

     <xsd:element name="publishURL" type="xsd:string"/>
     <xsd:element name="publishResponse" ref="xsd:string"/>
     ...
```

Fig. 5. Input/output parameters of the `searchData` and `publishData` operations

fer to the `searchData` operation, while the `publishURL` and `publishResponse` types refer to `publishData`. The search for a data set is performed through the `searchData` operation starting from a search string passed by the client. It contains the searching criteria expressed as attribute-value pairs regarding topic properties through which data sets are categorized within the system by means of appropriate metadata.

The outcome of the searching is a set of URLs (stored as an array of strings) pointing to the metadata of the data sets corresponding to the searching criteria. These kind of URLs are specifically targeted at the KDS service: it implements, in fact, a custom protocol for locating metadata descriptions of grid resources. A KDS URL has the form

kds://<hostname>/<metadataLocator>

and is able to unambiguously identify a metadata file.

Conversely, the `publishURL` is an URL on the file system of the client machine and points to the file containing the metadata about the data set to be published. This file is stored into the local KMR and a related entry is added to the registry of the KDS. As a result of the whole publishing operation a KDS URL is returned to the invoker through the `publishResponse`.

The parameters described above are used to declare the input and output SOAP messages of the search and publish operations provided by the DAS service. Pretty similar messages are defined also for the analogous KDS operations. Figure 6 reports the operations definition for the KDS service; where it is possible to note the practical usage of input and output messages.

4 Evaluation and Final Remarks

In the previous sections we discussed the design and implementation of the Knowledge Grid using WSRF-compliant services. This activity has been pre-

```
<portType name=KDSPortType"
      wsdlpp:extends="wsrpw:GetResourceProperty
          wsntw:NotificationProducer wsrlw:ImmediateResourceTermination"
          wsrp:ResourceProperties="tns:KDSResourceProperties">
          ...
          ...
          <operation name="searchResource">
                  <input message="tns:SearchInputMessage"/>
                  <output message="tns:SearchOutputMessage"/>
          </operation>

          <operation name="publishResource">
                  <input message="tns:PublishInputMessage"/>
                  <output message="tns:PublishOutputMessage"/>
          </operation>
          ...
          ...
</portType>
```

Fig. 6. KDS operations

ceded by a performance evaluation phase in which we analyzed the execution times of the WSRF Grid services for estimating the overhead introduced in the remote execution of data mining tasks on a Grid.

To evaluate the efficiency of the WSRF mechanisms discussed throughout the previous sections, we developed an experiment in which single WSRF-compliant K-Grid services executed the different steps described above for invoking the service, creating the resource, and accessing it. The deployed K-Grid service exported a service-specific operation named `clustering`, as well as the mandatory operations `createResource`, `subscribe` and `destroy`. In particular, the `clustering` operation was used to perform a data clustering analysis on a local data set using the expectation maximization (EM) algorithm. The K-Grid service and the client program have been developed using the WSRF Java library provided by Globus Toolkit 3.9.2. The service has been deployed on a machine in our Grid lab, while the client has been executed on remote and local machines connected to the Grid. The data set on which to apply the mining process contained 17000 instances (with a size of 5 MBytes) extracted from the *census* data set provided by the UCI repository [7].

After performing 20 independent experiments the execution times of the single steps have been measured. The experiments have been executed both within a LAN scenario and within a WAN network. The measurements showed that the data mining phase represents the 99.5% of the total execution time if client and service reside on a LAN, whereas the execution time on the WAN took about 88.3% of total time; the latter case included also the data-set download phase which accounted for about 10% of the total time. In both cases, the overhead due to specific WSRF mechanisms (resource creation, notification subscription, task submission, results notification) was very low with respect to the overall execution time; it accounted for an amount of time of about 0.5% and about 1.5% respectively.

In general, we can conclude that the overhead introduced by the WSRF mechanisms is marginal when the duration of the service-specific operations is long enough, as in typical data mining algorithms working on large data sets. Therefore, the WS-Resource Framework is suitable to be exploited for developing high-level services and distributed knowledge discovery applications on Grids.

Acknowledgement

This work has been partially supported by the European Commission FP6 Network of Excellence CoreGRID (Contract IST-2002-004265) and by the Italian MIUR FIRB Grid.it project RBNE01KNFP.

References

1. M. Cannataro and D. Talia. The Knowledge Grid. *Communitations of the ACM.* **46**(1):89–93, 2003.
2. The Global Grid Forum (GGF). http://www.ggf.org.
3. K. Czajkowski et al. The WS-Resource Framework Version 1.0. http://www-106.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf.
4. D. Box et al. Web Services Addressing (WS-Addressing), W3C Member Submission 10/8/04. http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810.
5. G. Bueti, A. Congiusta and D. Talia. Developing Distributed Data Mining Applications in the KNOWLEDGE GRID Framework. *Proc. VECPAR'04*. LNCS Series **3402**:156–169, 2004.
6. M. Cannataro, A. Congiusta, A. Pugliese, D. Talia and P. Trunfio. Distributed Data Mining on Grids: Services, Tools, and Applications. *IEEE Transactions on Systems, Man, and Cybernetics, Part B.* **34**(6):2451–2465, 2004.
7. The UCI Machine Learning Repository. http://www.ics.uci.edu/ mlearn/MLRepository.html.

Design and Development of a Core Grid Ontology ⋆

Wei Xing[1], **Marios D. Dikaiakos**[1],
Rizos Sakellariou[2], **Salvatore Orlando**[3], **Domenico Laforenza**[4]
[1] Department of Computer Science, University of Cyprus
[2] School of Computer Science, University of Manchester
[3] Dipartimento di Informatica, Universita Ca' Foscari di Venezia
[4] Information Science and Technologies Institute, Italian National Research Council
Email:{xing,mdd}@ucy.ac.cy, rizos@cs.man.ac.uk,
orlando@dsi.unive.it, domenico.laforenza@isti.cnr.it

Abstract. In this paper, we introduce a core Grid ontology that defines fundamental Grid domain concepts, vocabularies and relationships. This ontology is based on a general model of Grid infrastructures, and described in Web Ontology Language OWL. Such an ontology can play an important role in building Grid-related Knowledge base and in supporting the realization of the Semantic Grid.

1 Introduction

The Semantic Grid is perceived as an extension of current Grids in which information and services are given a well-defined meaning, better enabling computers and people to work in cooperation [5]. Ontologies are among the key building blocks for the semantic Grid. They determine the terms of Grid entities, resources, capabilities and the relationships between them, with which any kind of content can be *meaningful* by the addition of ontological annotations.

The main problem for building an ontology for Grids is that there is currently a multitude of proposed Grid architectures and Grid implementations, and these are comprised of thousands of Grid entities, services, components, and applications. It is thus very difficult, if at all feasible, to develop a complete Grid ontology that will include all aspects of Grids. Furthermore, different Grid sub-domains, such as Grid resource discovery and Grid job scheduling, normally have different views or interests of a Grid entity and its properties. This makes the definition of Grid entities and the relationships between them very hard.

To tackle these issues, we propose a core Grid ontology (CGO) that defines fundamental Grid-specific concepts, vocabularies and relationships, based on a general model for Grids. This ontology can provide a common basis for representing Grid knowledge about Grid resources, Grid middleware, services, applications, and Grid users. A key challenge that needs to be addressed in this context is to make the core Grid ontology extensible and general enough to be used by, or incorporated in, different Grid systems and tools. To address

⋆ Work supported in part by the European Commission under the CoreGRID project.

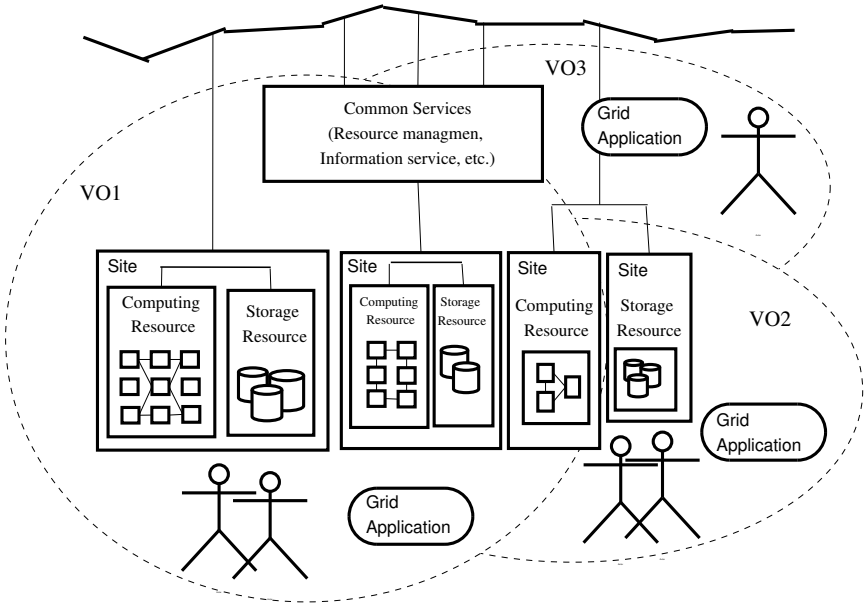


Fig. 1. The Overview of the Proposed Grid Model

this challenge, we design the CGO so that it represents a general model for Grids, which is compatible to major Grid infrastructures [14, 15, 1, 10, 4]. Also by adopting the Resource Description Framework (RDF) graph data model [9], a W3C recommendation, as the data model of choice for representing the CGO concepts and their relationships. The characteristics of the RDF data model make the ontology easier to extend by either adding new classes in it or adding extra properties (slots) into a defined class without any conflict with existing definitions.

The remaining of this paper is organized as follows. In Section 2, we introduce the proposed Grid model. In Section 3, we present an overview of the Core Grid Ontology. We illustrate how to use the Core Grid Ontology with an example in Section 4. Finally, we conclude our paper in Section 5.

2 Grid Model Description

The Grid is a platform for coordinated resource sharing and problem solving in dynamic, multi-institutional Virtual Organizations (VO) [7]. In essence, the Grid can be considered as a collection of Virtual Organizations and different kinds of resources. Resources are combined and organized by Grid middleware to provide Grid users with computing power, storage capability, and services, required for problem solving. VOs enable disparate groups of organizations and/or individuals to share resources in a controlled fashion, so that members may collaborate to achieve a shared goal. In our proposed Grid model, we view the Grid as a constellation of Virtual Organizations (VOs), which includes VOs, users, applications, middleware, services, computing and storage resources, networks, policies of use (see Figure1).

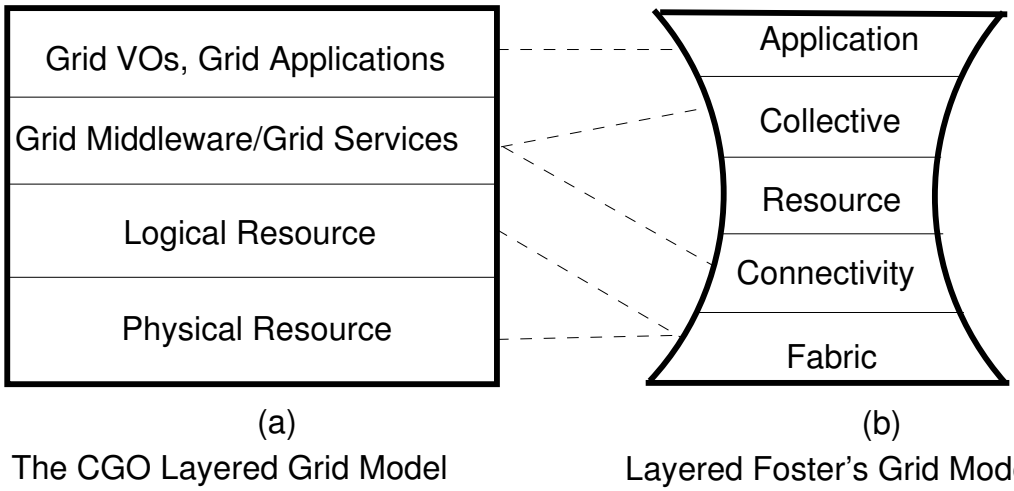


Fig. 2. The Layered Grid Model

One design issue of the core Grid ontology is to capture a "right" model for the Grid, that could be used to further specify Grid concepts, vocabularies, relations, and constrains. This model must remain simple and should have a proper level of abstraction that hides the numerous details involved in Grids. It should also provide a general view of important aspects of Grids [6].

The proposed model is actually a layer-structured model. As shown in Figure 2, the top layer includes multi-VOs, Grid Users, and Applications; Grid middleware and Grid services lie on the second layer; third layer is the VO-based "virtual" resources layer; the "real" physical Grid resources, such as clusters, networks, is at the bottom. Compared with the Foster's layered Grid model in [7], our layered Grid model is designed around a simple four-layer scheme, it combines the features of some adjacent layers of Foster's model and splits other layers apart [7]. In the top layer of our Grid model, we add Grid VOs and users besides applications. From our point of view, the VOs are actually a container that has users and applications in it. Thus, VO should be a fundamental element of Grids. The layer of the Grid middleware and Grid services of our model can be mapped into the three layers of the Foster's model, e.g., Connectivity, Resource, and Collective layer. The two layers, e.g. Logical Resource and Physical Resource, are correspondent with the fabric layer of the Foster's model. The intention of our proposed model is to provide a general, abstract view of Grids, which avoids any specific architectures or functionalities. The results of our division is a more general, extensible, open, VO-oriented model.

One notable aspect of the proposed model is that we distinguish Grid resources as logical or physical. A physical resource (PR) is a "real" resource that is part of a Grid, and the logical resource (LR) is "virtual" resource that a VO controls according to its policies, rules, and availability. Grid middleware and Grid services are responsible for mapping LR into PR. From the view of VOs and Grid Applications, the LR is more "realistic" than the PR, as Grids are VO-oriented. Since the Grid resources normally serve multi-VOs concurrently, the LR are many more in absolute numbers than the PR.

3 A Core Grid Ontology

The core Grid ontology is designed to represent Grid knowledge of Grid systems. Therefore, it should be open and extensible as there are thousands of Grid entities, services, components, and applications of different Grid architectures and Grid implementations. To cope with the openness and extensibility requirements, we adopt the Web Ontology Language OWL to describe the terms and classes in the core Grid ontology [13]. OWL is a semantic markup language for publishing and sharing ontologies on the World Wide Web, which is developed as a vocabulary extension of the Resource Description Framework (RDF) [3]. Given the fact that both RDF and OWL are W3C recommendations, the core Grid ontology is thus open, and compatible with other systems. Furthermore, the characteristics of the RDF data model make the ontology easier to extend by either adding new classes in the ontology or adding extra properties (or slots) into a defined class without any conflict with existing definitions. The RDF data model is a directed graph with labeled nodes and arcs; the arcs are directed from one node (i.e. subject) to another node (i.e. object). The object may be linked to other nodes (e.g. other new classes) through a property. One key feature of this data model is that properties in RDF are defined globally, that is, they are not encapsulated as attributes in class definitions. It is thus possible to define new properties that apply to an existing class without changing that class.

One main challenge in developing a core Grid ontology is to provide formal definitions and axioms that constrain the interpretation of classes and terms. In the following sections, we describe the concepts and, in particular, represent their constraints on the Grid domain according to the knowledge derived from analyzing, evaluating, and experimenting with different Grid architectures, production middleware and large Grid infrastructures, such as, Globus, Unicore, DataGrid, Crossgrid, and EGEE [14, 15, 1, 10, 4]. We first define a set of core Grid ontology classes, which reflect on all the elements of the abstract model. Subclasses can then be added accordingly. After that, we define the relationships and constraints among the ontology classes using RDF properties. These properties are links among the defined classes. Finally, the classes, properties, and instances together form a knowledge base that captures the configuration and state of specific Grid infrastructures. Such a knowledge-base can be used for various inquiries and for decision-support of end-users, application developers, Grid administrators, etc.

3.1 The Core Grid Ontology Classes and the Properties

In the proposed Grid model, we view the Grid as a collection of VOs, any amount of Grid resources, Grid Middleware and Services. Thus we can group the concepts (or Grid Objects) of the Core Grid Ontology into three categories:

1. VO-related: the classes that reflect the Grid entities on the top level of the proposed Grid model, including VO, GridUser, GridApplication, and Policy.

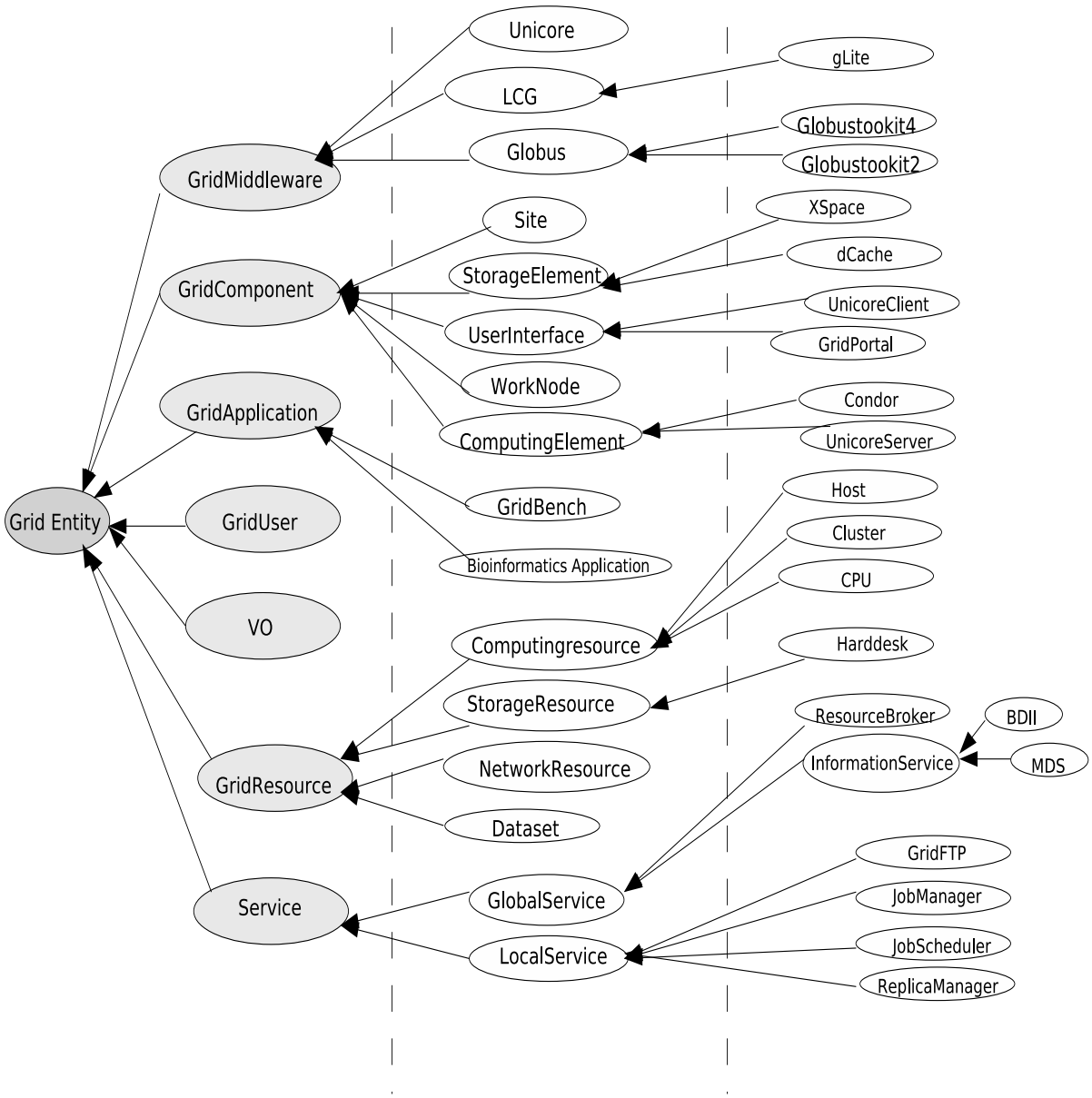


Fig. 3. The Overview of the Core Grid Ontology Classes

2. Grid Resource: Any resources in/on the Grid, including Computing Resource, Storage Resource, Network Resource, and Dataset. The classes of this catalog represent the Grid entities of the logical resource layer and physical resource layer of the Grid model.
3. Grid Middleware & Service: Grid middleware, functions, components, and services, which provide mechanisms and functionalities that provide the Grid Resource to server VOs.

Based on the proposed Grid model, the following types of the core classes are included in the CGO collection: VO, GridResource, GridMiddleware, GridComponent, GridUser, GridApplication, Service. A overview of the CGO classes is given in Figure 3. The meaning of the top level classes is presented as follows:

– VO: A set of individuals and/or institutions defined by sharing policies and rules form what we call a virtual organization (VO).

- GridResource: Any Hardware, Software, computing resource, storage resource, network resource within a Grid.
- GridMiddleware: Software that glue Grid services together following a specific Grid architecture.
- GridComponent: An architecturally significant part of a Grid system with well-defined inbound and outbound interfaces and associated protocols that encapsulate a cohesive documented set of responsibilities.
- GridUser: Users who use the Grid system.
- GridApplication: Applications that can run on Grids.
- Service: Any services that can provide one or more Grid functionalities.

The core classes (see in Figure 3) are fundamental concepts, elements, and aspects of a Grid system. They provide a "framework" for representing any entity of a Grid system. In other words, all important Grid entities and resources in/on Grids can be "located" within this framework. For example, to describe a DataGrid-based Grid component ComputingElement (CE), which is responsible for providing computing power and comprised of one or more similar machines managed by a single scheduler/job queue, following aspects need to be determined [1]:

- Which VOs does it support?
- What kinds of Service run on it?
- Which GridMiddleware has been installed?
- Which kinds of applications can it run?
- How many Grid resources can it provide?

Therefore, we can represent the Data Grid CE using the defined core ontology classes in the RDF triple model as follows:

{ CE *isA* *ComputingElement*};
{ CE *supportVO* *VO*};
{ CE *runningService* *Service*};
{ CE *hasInstalled* *GridMiddleware*};
{ CE *totalCPU* XMLSchema#int };
{ CE *freeCPU* XMLSchema#int }.

In order to represent the relationships and constrains among the ontology classes, we define the properties that provide the semantic meaning for the Core Grid Ontology entities. As we mentioned earlier, one key design goal of the CGO is to be open and extensible. Hence users can extend the CGO by adding their classes and properties on a "read-to-have" basis. In other words, we intend to provide a "framework" for representing a Grid system instead of having a whole, complete set of classes and properties for a Grid system. In this paper, we only present the core properties that reflect the relationships among the core ontology classes concerning with the class ComputingResource to illustrate how the relationships among the classes are defined (or linked). Currently, the whole defined properties can be founded in [12]. Table 1 shows the properties related with Computing Resource in the core Grid ontology.

Properties	*Description*
supportVO	*belong to VO*
runningService	*Grid service running on*
hasName	*Name of any resource*
coService	*Services related to the running services*
accessControlBaseType	*Policy for accessing Grid resources*
maxRunningJobs	*maxim number of job in one queue*
operatingSystem	*type of OS on host*
runningEnvironment	*middleware or services environment*
storageDevice	*interface of the storage element*
fileSystem	*the type of the file system on the storage element*
totalCPU	*the number of the total CPUs*
freeCPU	*the number of the available CPUs*

Table 1. The Properties related with CE in the Core Grid Ontology

3.2 Representing a Grid Entity using the Core Grid Ontology

We adopted OWL to describe the identified Grid entities, not only defining the concepts of them but also the relationships and constrains among them. For example, we defined the ComputingElement (CE) as:

- *CE is a Grid entity has several kinds of computing resources, such as CPU, Memory. The resource is organized by grid middleware and use Grid services to provide computing power to VO(s).*
- *A CE is comprised of one or more similar machines managed by a single scheduler/job queue.*

According to the definition, we describe the constrains of the class ComputingElement: a) a CE must support at least one VO; b) a CE machine must run a GridManager service; c) a CE machine must run a JobManager service and JobScheduler service. The three restrictions can be described in Description Logics as follows [2]:

$$\begin{aligned} \text{CE} \sqsubseteq\ & \exists\ \text{supportVO}\ \ \text{VO}. \\ & \exists\ \text{runningSevice (GridManager)}, \\ & \exists\ \text{runningSevice (JobManager} \sqcup \text{JobScheduler)}. \end{aligned}$$

The above formula represents the necessary conditions of a Computing Element:

(1) isA Computing Resource.
(2) at least one of the values of the runningService property is of type GridManager.
(3) at least one of the values of the runningService property are the union of
    - JobManager
    - JobScheduler
(4) at least one of the values of supportVO property is of type VO.

We thus describe the CE from the following facts: i) There is at least one VO supported; ii) There are a number of CPUs this CE dedicates to its VO; iii) There are some specified services running on it; iv) The GridMiddleware it installed; v) It has some worker nodes. The Table 2 shows the CE description in OWL using the Core Grid Ontology.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.owl-ontologies.com/unnamed.owl#"
  xml:base="http://www.owl-ontologies.com/unnamed.owl">
.....

<owl:Class rdf:ID="ComputingElement">
     <rdfs:subClassOf>
       <owl:Restriction>
         <owl:someValuesFrom>
           <owl:Class>
             <owl:unionOf rdf:parseType="Collection">
               <owl:Class rdf:about="#Jobmanager"/>
               <owl:Class rdf:about="#JobScheduler"/>
             </owl:unionOf>
           </owl:Class>
         </owl:someValuesFrom>
         <owl:onProperty rdf:resource="#runningSevice"/>
       </owl:Restriction>
     </rdfs:subClassOf>
     <rdfs:subClassOf rdf:resource="#EDG"/>
     <rdfs:subClassOf>
       <owl:Restriction>
         <owl:onProperty rdf:resource="#runningSevice"/>
         <owl:someValuesFrom rdf:resource="#Gridmanager"/>
       </owl:Restriction>
     </rdfs:subClassOf>
  </owl:Class>
......
```

Table 2. CE class Definition in Core Grid Ontology

4 Example: Generating the Ontology Instances

After introducing the Core Grid Ontology, we represent a ComputingElement (CE) of the CY01-LCG2 Grid node of the EGEE infrastructure to show how the ontology can help generate the knowledge-based information about a CE [4].

In the CY01-LCG2 Grid node, we have a CE named ce101.grid.ucy.ac.cy. We describe the ce101 instance based on the definition of the ComputingElement class. In CGO, it is defined that there must be three services running on a CE. Furthermore, since the *Gridmiddleware* of the CY01-LCG2 Grid node is *LCG*, we then can "*deduce*" that the jobmanager service is *openPBS* and job scheduler service is *MAUI*. So we can represent the CE as:

1. ce101.grid.ucy.ac.cy is a LCG-based CE;
2. It supports BioMed VO;
3. It has LCG-2.6.0 installed;
4. It runs globus-gatekeeper as Grid manager;
5. It runs openPBS as JobManager, and MAUI as Jobscheduler;
6. It has 20 WorkerNodes, and provides 40 CPUs.

By this way, we can represent an instance of a CE (i.e., ce101.grid.ucy.ac.cy) in RDF/OWL. It is in fact the knowledge about ce101.grid.ucy.ac.cy, which can be queried by Grid users, Grid services, or Grid applications. Figure 4 shows the ontology definition and the OWL description of the example, presented as an OWL graph.

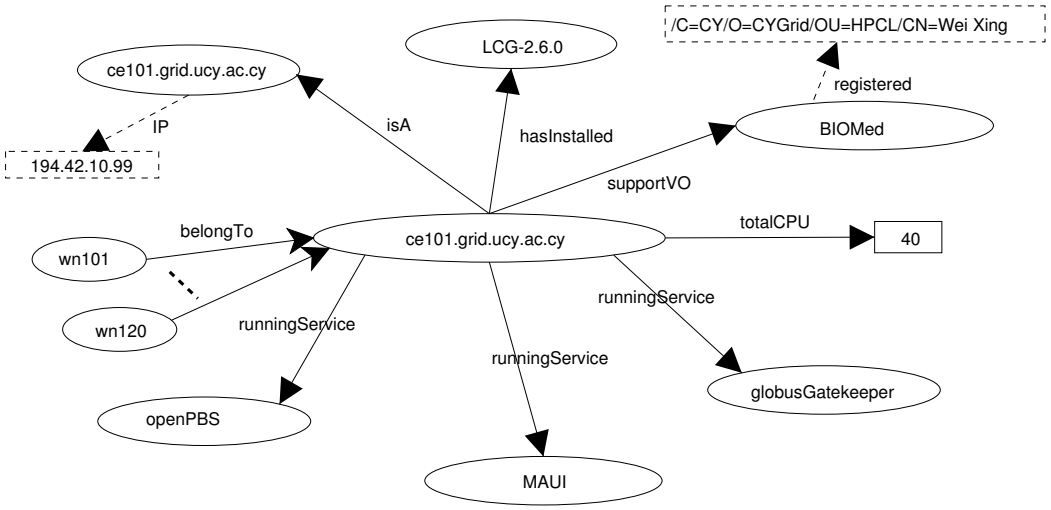


Fig. 4. The Core Grid Ontology Instance

5 Conclusions and Future Work

In this paper, we present our initial work on building a core Grid ontology. We first introduce an abstract model of Grid. After that, we design and develop a core Grid ontology that expresses the basic concepts and relationships of Grid entities and Grid resources according to the proposed Grid model. The flexibility and extensibility of the ontology allows it to be used, among other things, for Grid information integration, information searching, resource discovery and resource allocation management. Additionally, the fact that it is Grid architecture and implementation independent, renders it quite useful for hybrid large-scale Grids.

In the future, we plan to extend our work in the following directions:

1 Developing a toll that can automatically generate the instances of the CGO using Jena [8]. Currently we generate the instances of the CGO manually using Protege [11]. It is obviously not suit for the Grid system, which includes thousands of instances. Therefore, we need a tool that can automatically generate and update the instances dynamically.
2 Support knowledge-based queries. The ontologies/knowledge will be stored in a distributed manner. We need a suitable OWL query language and distributed query mechanism to query those distributed knowledge efficiently.

3 Knowledge-based resource discovery. Traditionally, the resource discovery is performed by key-word based one-by-one match mechanism. With the Core Grid Ontology, one can search the required resources with a knowledge-based search. Namely, good selection strategies, which use various kinds of knowledge, are used to guide the resource discovery. It will not only improve the performance of the resource discovery, also improve the quality of the results.

6 Acknowledgments

We thank George Tsouloupas,Nick Drummond, Matthew Horridge, Robert Stevens and Hai Wang for helpful discussions.

References

1. http://eu-datagrid.web.cern.ch/eu-datagrid/.
2. *The Description Logic Handbook*. Cambridge University Press, 2003.
3. D. Brickley and R.V. Guha (editors). RDF Vocabulary Description Language 1.0: RDF Schema. W3C Working Draft, October 2003. http://www.w3.org/TR/rdf-schema/.
4. S. Campana and A. Sciaba M. Litmaath. LCG-2 Middleware Overview. LCG Technical Document. https://edms.cern.ch/file/498079//LCG-mw.pdf.
5. N.R.Shadbolt De Roure, D.Jennings, editor. *The Semantic Grid: Past, Present and Future*. IEEE, March 2005.
6. M. Dikaiakos and A. Artemiou. Navigating the grid information space: Design and implementation of the ovid browser. Technical Report Technical Report TR-2004-07, Department of Computer Science, University of Cyprus, December 2004.
7. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Supercomputer Applications*, 2001.
8. http://jena.sourceforge.net/ontology/. *Jena 2 Ontology API*.
9. F. Manola and E. Miller (editors). RDF Primer. W3C Working Draft, October 2003. http://www.w3.org/TR/rdf-primer/.
10. J. Marco, W. Xing R. Marco, and M. Dikaiakos et al. First prototype of the crossgrid testbed. volume vol. 2970 of *Lecture Notes in Computer Science series*, pages 67–77. Springer, Santiago de Compostela, Spain, February 2003.
11. N. F. Noy, S. Decker M. Sintek, R. W. Fergerson M. Crubezy, and M. A. Musen. Creating semantic web contents with protege-2000. *IEEE Intelligent Systems*, 16:60–71, 2001.
12. http://grid.ucy.ac.cy/grisen/coreonto.owl.
13. P.F. Patel-Schneider, P. Hayes, and I. Horrocks. *OWL Web Ontology Language Semantics and Abstract Syntax*. World Wide Web Consortium, February 2004.
14. S. Tuecke, I. Foster K. Czajkowski, C. Kesselman J. Frey, S. Graham, T. Sandholm T. Maguire, and D. Snelling P. Vanderbilt. Open Grid Service Infrastructure (OGSI) version 1.0. *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.
15. Ph. Wieder and D. Mallmann. UniGrids - Uniform Interface to Grid Services. In *7th HLRS Metacomputing and Grid Workshop*. Stuttgart, Germany, April 2004.

Towards a common deployment model for Grid systems

Massimo Coppola[1], Marco Danelutto[2], Sébastien Lacour[3], Christian Pérez[3], Thierry Priol[3], Nicola Tonellotto[1,4], and Corrado Zoccolo[2]

[1] ISTI, Area della Ricerca CNR, 56124 Pisa, Italy
[2] Dept. of Computer Science, University of Pisa, L.go B. Pontecorvo, 3, 56127 Pisa, Italy
[3] IRISA/INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
[4] Information Engineering Dept., University of Pisa, Via G. Caruso 16, 56122 Italy

Abstract. Deploying applications within a Grid infrastructure is an important aspect that has not yet been fully addressed. This is particularly true when high-level abstractions, like objects or components, are offered to the programmers. High-level applications are built on run-time supports that require the deployment process to span over and coordinate several middleware systems, in an application independent way. This paper addresses deployment by illustrating how it has been handled within two projects (ASSIST and GridCCM). As the result of the integration of the experience gained by researchers involved in these two projects, a common deployment process is presented.

1 Introduction

The Grid vision introduced in the end of the nineties has now become a reality with the availability of quite a few Grid infrastructures, most of them experimental but some others will come soon in production. Although most of the research and development efforts have been spent in the design of Grid middleware systems (i.e., the Grid operating systems), the question of how to program such large scale computing infrastructures remains open. Programming such computing infrastructures will be quite complex considering its parallel and distributed nature. The programmer vision of a Grid infrastructure is often determined by its programming model. The level of abstraction that is proposed today is rather low, giving the vision either of a parallel machine, with a message-passing layer such as MPI, or a distributed system with a set of services, such as Web Services, to be orchestrated. Both of these two approaches offer a very low level programming abstraction and are not really adequate, limiting the spectrum of applications that could take benefit from Grid infrastructures. Of course such approaches may be sufficient for simple applications but a Grid infrastructure has to be generic enough to be able to handle both simple and complex applications. To overcome this situation, it is required to propose high level abstractions to facilitate the programming of Grid infrastructures and in a longer term to be able to develop more secure and robust

next generation Grid middleware systems by using these high level abstractions for their design as well. Thus, the current situation is very similar to what happened with computers in the sixties: minimalist operating systems were developed first with assembly languages before being developed in the seventies by languages that offer higher levels of abstraction.

Several research groups are already investigating how to design or adapt programming models that provide this required level of abstraction. Among these models, component-oriented programming models are good candidates to deal with the complexity of programming Grid infrastructures. A Grid application can be seen as a collection of components interconnected in a certain way that must be deployed on available computing resources managed by the Grid infrastructure. Components can be reused for new Grid applications, reducing the time to build new applications. However, from our experience such models have to be combined with other programming models that are required within a Grid infrastructure. It is imaginable that a parallel program can be encapsulated within a component. Such a parallel program is based on a parallel programming model which might be for instance message-based or skeleton-based. Moreover, a component oriented programming model can be coupled with a service oriented approach exposing some component ports as services through the use of Web Services.

The results of this is that this combination of several models to design Grid applications leads to a major challenge: the deployment of applications within a Grid infrastructure. Such programming models are always implemented through various runtime or middleware systems that have their own dependencies vis-à-vis of operating systems, making it extremely challenging to deploy applications within a heterogeneous environment, which is an intrinsic property of a Grid infrastructure.

The objective of this paper is to propose a common deployment process based on the experience gained from the ASSIST and GridCCM projects. This paper is organized as follows. Section 2 gives a short description of the ASSIST and GridCCM projects. Section 3 describes our common analysis of what should be the different steps to deploy grid applications. Section 4 gives a short description of GEA and ADAGE, the two deployment systems designed respectively for ASSIST and GridCCM, and how they already conform to the common model. Finally, Sect. 5 concludes the paper and presents some perspectives.

2 ASSIST and GridCCM Software Component Models

Both University of Pisa and INRIA-Rennes have investigated the problem of deploying component-based Grid applications in the context of the ASSIST and GridCCM programming environments and came out with two approaches with some similarities and differences. In the framework of the CoreGRID Network of Excellence, the two research groups decided to join their efforts to develop a common deployment process suitable for both projects taking benefits of the experience of both groups. In the remaining part of this section, the AS-

SIST and GridCCM programming and component models are presented, so as to illustrate the common requirements on the deployment system.

2.1 Assist

ASSIST (A Software development System based upon Integrated Skeleton Technology) is a complete programming environment [10] aimed at the efficient development (w.r.t. development complexity and overall performance) of high-performance multi-disciplinary applications, at managing their complexity and lowering their time to market. To achieve this goal in a Grid Computing setting, the environment must deal with heterogeneous resources, and with dynamic changes in the computational behavior of resources and application modules.

ASSIST provides an efficient and high-performance way of expressing the parallel computational cores, and the Grid.it component [1] support allows for integration of different component frameworks, and for implementation of automatic component adaptation to varying resource performance or varying computational needs of the program.

The deployment of an ASSIST application or Grid.it component is a complex task, involving the selection of resources and configuration of different set of processes in such a way that they are able to cooperate, obtaining high-performance. Moreover, this task does not finish when an application is launched: an application component can request more resources, in order to adapt its configuration to fulfill specified performance requirements [3].

ASSIST components have a platform independent description encoded in ALDL [4], an XML dialect expressing the detailed requirements and execution constraints of the elementary blocks composing the component. It is interpreted by the GEA tool (see Sect. 4.1).

2.2 GridCCM: a Parallel Component Model

GridCCM [9] is a research prototype that targets scientific code coupling applications. Its programming model extends the CORBA Component Model (CCM) with the concept of parallel components. A parallel component, defined as a collection of sequential components, aims at embedding a parallel code whatever the parallelism technology is (MPI, PVM, OpenMP, ...). Interactions between parallel components are handled by GridCCM which supports optimized scheduled MxN communications. MxN data redistributions and communication scheduling may be extended at user-level.

The deployment of a GridCCM application turns out to be a complex task because several middleware systems may be involved. There are the component middleware, which implies to deploy CCM applications, and the technology used by the parallel component which may be MPI, PVM or OpenMP for example. Moreover, to deal with network issues, an environment like PadicoTM [5] should be also deployed with the application.

The description of an GridCCM application is achieved thanks to an extension of the XML CCM Component Software Description language. Hence, a

parallel component has a parallel implementation whose description may be a MPI description [7]. It should be the only input required by a deployment tool to a user as show in Sect. 4.2.

2.3 Discussion

Both ASSIST and GridCCM expose programming models that required advanced deployment tools to efficiently handle the different elements of an application to be deployed. Moreover, they provide distinct features like the dynamic behavior and the different Grid middleware support of ASSIST and the multi-middleware application support of GridCCM. Hence, a common deployment process will help in integrating features needed for their deployment.

3 General Overview of the Deployment Process

Starting from a description of an application and a user objective function, the deployment process is responsible for automatically performing all the steps needed to start the execution of the application on a set of selected resources. This is done in order to avoid the user from directly dealing with heterogeneous resource management mechanisms.

From the point of view of the execution, a component contains a structured set of binary executables and requirements for their instantiation. Our objectives include generating deployment plans

- to deploy components in a multi-middleware environment,
- to dynamically alter a previous configuration, adding new computational resources to a running application,
- for re-deployment, when a complete restart from a previous checkpoint is needed (severe performance degradation or failure of several resources).

A framework for the automatic execution of applications can be composed of several interacting entities in charge of distinct activities, as depicted in Fig. 1. The logical order of the activities is fixed (Submission, Discovery, Selection, Planning, Enactment, Execution). Some steps have to be re-executed when the application configuration is changed at run-time. Moreover, the steps in the grey box, that interact closely, can be iterated until a suitable set of resources is found.

In the following we describe the activities involved in the deployment of an application on a Grid. We also detail the inputs that must be provided by the user or the deployment framework to perform such activities.

3.1 Application Submission

This is the only activity which the user must be involved in, to provide the information necessary to drive the following phases. This information is provided through a file containing a description of the components of the application, of their interactions, and of the required resource characteristics.

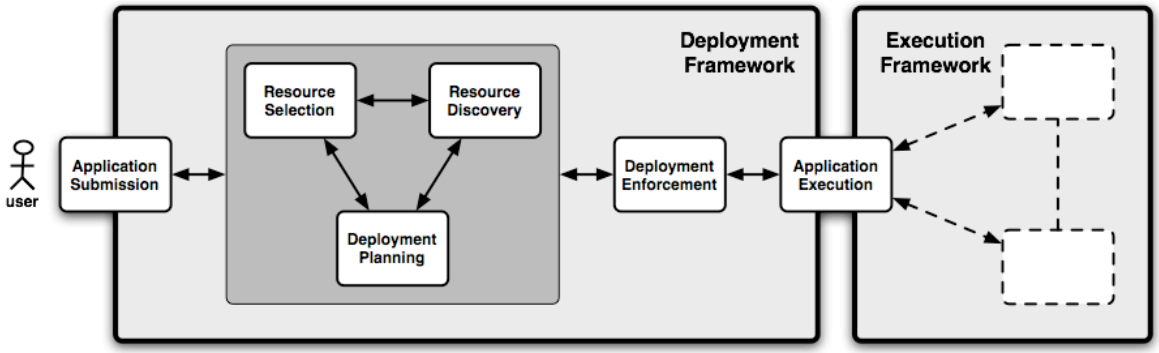


Fig. 1. Activities involved in the deployment process of an application.

Application Description. The description of (the components of) the submitted application, written in an user-understandable specification language, is composed of various kinds of data.

- Module description: the executable files, I/O data and configuration files which make up each module (e.g. each process).
- Information to guide the stages related to mapping the application onto resources,
  - resource constraints: the characteristics that Grid resources (computational, storage and network) must possess to execute the application,
  - execution platform constraints: the software (libraries, middleware systems) that must be installed to satisfy application dependencies,
  - placement policies: restrictions or hints for the placement of subsets of application processes (e.g. co-location, location within a specific network domain, or network performance requirements, etc.),
  - resource ranking: an objective function provided by the user, stating the optimization goal of application mapping. Resource ranking is exploited to select the best resource, or set of them, among those satisfying the given requirements for a single application process.

  Resource constraints can be expressed as *unitary requirements*, that is requirements that must be respected by a single module or resource (e.g. CPU rating), and as *aggregate requirements*, i.e., requirements that a set of resources or a module group must respect at the same time (e.g. all the resources on the same LAN, access to a shared file system); some placement policies are implicitly aggregate requirements.
- Deployment directives: they determine the tasks that must be performed to set up the application runtime environment, and to start the actual execution.

As discussed in the following sections, the provided information is used throughout the deployment process.

3.2 Resource Discovery

This activity is aimed at finding the resources compatible with the execution of the application. In the application description several requirements can be specified that available resources must respect to be eligible for execution. The requirements can specify hardware characteristics (e.g. CPU rating, available memory, disk space), software ones (e.g. OS, libraries, compilers, runtime environments), services needed to deploy components (e.g. accessible TCP ports, specific file transfer protocols), and particular execution services (e.g. to configure the application execution environment).

Resources satisfying unitary requirements can be discovered, interacting with Grid Information Services. Then, the information needed to perform resource selection (that considers also aggregate requirements), must be collected, for each suitable resource found.

The GIS[5] can be composed of various software systems, implementing information providers that communicate with different protocols (MDS-2, MDS-3, MDS-4, NWS, iGrid, custom). Some of these systems provide only static information, while others can report dynamic information about resource state and performance, including network topology and characteristics. In order to interact with such different entities, an intermediate translation layer between the requirements needed by the user and the information provided is necessary. Information retrieved from different sources is mapped to a standard schema for resource description that can be exploited in the following activities independently from the information source.

3.3 Resource Selection

When information about available resources is collected, the proper resources that will host the execution of the application must be selected, and the different parts of each component have to be mapped on some of the selected resources. This activity also implies satisfying all the aggregate requirements within the application. Thus, repeated interaction with the resource discovery mechanisms may be needed to find the best set of resources, also exploiting dynamic information.

At this point, the user objective function must be evaluated against the characteristics and available services of the resources (expressed in the normalized resource description schema), establishing a resource ranking where appropriate in order to find a suitable solution.

3.4 Deployment Planning

A component-based application can require different services installed on the selected resources to host its execution. Moreover, additional services can be transferred/activated on the resources or configured to set up the hosting environment.

[5] Grid Information Service

Each of these ancillary applications has a well-defined deployment schema, that describes the workflow of actions needed to set up the hosting environment before the actual execution can start.

After resource selection, an abstract deployment plan is computed by gathering the deployment schemata of all application modules. The abstract plan is then mapped on the resources, and turned into a concrete plan, identifying all the services and protocols that will be exploited in the next phase on each resource, in order to set up and start the runtime environment of the application.

For example, to transfer files we must select a protocol (e.g. HTTP, GridFTP), start or configure the related services and resources, and finally start the transfer. At the end of this phase, the concrete deployment plan must be generated, specifying every single task to perform to deploy the application.

This activity can require repeated interactions with the resource discovery and selection phases because some problems about the transformation from the deployment schema to the deployment plan can arise, thus the elimination of one or more eligible resources can force to find new resources, and restart the whole planning process.

3.5 Deployment Enactment

The concrete deployment plan developed in the previous phase is submitted to the execution framework, which is in charge of the execution of the tasks needed to deploy the application. This service must ensure a correct execution of the deployment tasks while respecting the precedences described in the deployment plan. At the end of this phase, the execution environment of the application must be ready to start the actual execution.

This activity must deal with different kinds of software and middleware systems; the selection of the right ones depends on the concrete deployment plan. The implementation of the services that will perform this activity must be flexible enough to implement the functionalities to interact with different services, as well as to add mechanisms to deal with new services.

Changes in the state of the resources can force a new deployment plan for some tasks. Hence, this phase can require interactions with the previous one.

3.6 Application Execution

The deployment process for adaptive Grid applications does not finish when the application is started. Several activities have to be performed while the application is active, and actually the deployment system must rely on at least one permanent process or daemon. The whole application life-cycle must be managed, in order to support new resource requests for application adaptation, to schedule a restart if an application failure is detected, and to release resources when the normal termination is reached. These monitoring and controlling activities have to be mediated by the deployment support (actual mechanisms depend on the middleware), and it does seem possible to reliably perform them over noisy, low-bandwidth or mobile networks.

4 Current Prototypes

4.1 GEA

The Grid Execution Agent (GEA, [4]) is an automatic tool developed within the *Grid.it* project to seamlessly run complex component-based Grid applications on different grid middleware systems. The application classes targeted are mainly high-performance, data/computation intensive, and distributed ones.

GEA is an open framework implementing several interfaces; these interfaces provide the functionalities needed to effectively deploy on a Grid complex applications, developed with high-level programming tools like ASSIST. GEA uses a description of the application in a general format (ALDL [4]), encoding the resource requirements for all kinds of applications and execution architectures. GEA, starting from the ALDL description, automatically performs resource discovery and selection, handles data and executable file transfers. It plans and enacts the deployment workflow of ASSIST components, starting in the proper order the server processes needed by any component as well as the application processes. As a result of the deployment process, different parts of the ALDL description are translated into the appropriate forms for the middleware used on each part of the application.

In detail, the Grid Execution Agent provides the following capabilities:

- Virtualization of several basic functions w.r.t. the underlying middleware systems (see Tab. 1), by means of the Grid Abstract Machine (GAM) [2,4]. The GAM level provides an abstraction of security mechanisms, resource discovery, resource selection, (secure) data & code staging and execution.
- Resource location (by static configuration or Globus MDS) and monitoring (only through NWS) during the whole deployment and execution phases.
- Instantiation and termination of ASSIST components, managing support processes belonging to different middleware systems according to the ASSIST component deployment workflow.
- Automatic reconfiguration of components (GEA provides an API to the application runtime to dynamically request new resources).

4.2 ADAGE

ADAGE [6] (*Automatic Deployment of Applications in a Grid Environment*) is a research project that aims at studying the deployment issues related to multi-middleware applications. One of its originality is to use a generic application description model (GADe) [8] to be able to handle several middleware systems. ADAGE follows the deployment process described in this paper.

With respect to application submission, ADAGE requires an application description, which is specific to a programming model, a reference to a resource information service (MDS2, or an XML file), and a control parameter file. The application description is internally translated into a generic description so as to support multi-middleware applications. The control parameter file allows a

Table 1. Features of the common deployment process supported by GEA and ADAGE.

Feature	GEA	ADAGE
Component description in input	ALDL (generic)	Many, via GADe[a]
Multi-middleware application	Yes (in progress)	Yes
Dynamic application	Yes	No
Resource constraints	Yes	Yes
Execution constraints	Yes	Yes
Grid Middleware	Many, via GAM[b]	SSH and GT2

[a] MPI, CCM, GridCCM, JXTA, etc.
[b] GAM supports Globus toolkit 2,3 and SSH; GT4 support is under development.

user to express constraints on the placement policies which are specific to an execution. For example, a constraint may affect the latency and bandwidth between a computational component and a visualization component. However, the two implemented schedulers, random and round-robin, does not take into account any control parameters but the constraints of the submission method (GT2 or SSH). Processor architecture and operating system constraints are taking into account.

The support of multi-middleware applications is based on a plugin mechanism. The plugin is involved in the conversion from the specific to the generic application description but also during the execution phase so as to deal with specific middleware configuration actions.

ADAGE currently deploys only static applications. It supports standard programming models like MPI (MPICH1-P4 and MPICH-G2), CCM and JXTA, as well as more advanced programming models like GridCCM. The current support of GridCCM is restricted to MPI-based parallel components.

4.3 Comparison of GEA and ADAGE

Table 1 sums up the similarities and difference between GEA and ADAGE with respect to the features of our common deployment process. The two prototypes are different approximations of the general model: GEA supports dynamic ASSIST applications. Dynamicity, instead, is not currently supported by ADAGE. On the other hand, multi-middleware applications are fully supported in ADAGE, as it is a fundamental requirement of GridCCM. Its support in GEA is in progress, following the incorporation of those middleware systems in the ASSIST component framework.

5 Conclusion

ASSIST and GridCCM programming models requires advanced deployment tools to handle both application and grid complexity. This paper has presented a common deployment process for components within a Grid infrastructure. This model is the result of several visits and meetings that was held during the last past months. It suits well the needs of the two projects, with respect to the

support of heterogeneous hardware and middleware, and of dynamic reconfiguration. The current implementations of the two deployment systems – GEA and ADAGE– share a common subset of features represented in the deployment process. Each prototype implements some of the more advanced features. This motivates the prosecution of the collaboration.

Next steps in the collaboration will focus on the extension of each existing prototype by integrating the useful features present in the other: dynamicity in ADAGE and extending multi-middleware support in GEA. Another topic of collaboration is the definition of a common API for resource discovery, and a common schema for resource description.

References

1. M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in the Grid.it project. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications (June 2004, Saint Malo, France)*. Springer, January 2005.
2. M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, Jan. 2006.
3. M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, LNCS, pages 771–781, Lisboa, Portugal, August 2005. Springer.
4. M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. HPC Application Execution on Grids. In V. Getov, D. Laforenza, and A. Reinefeld, editors, *Future Generation Grids*, CoreGrid series. Springer, 2006. Dagstuhl Seminar 04451 – November 2004.
5. A. Denis, C. Pérez, and T. Priol. PadicoTM: An open integration framework for communication middleware and runtimes. *Future Generation Computer Systems*, 19(4):575–585, May 2003.
6. S. Lacour, C. Pérez, and T. Priol. A software architecture for automatic deployment of CORBA components using grid technologies. In *Proceedings of the 1st Francophone Conference On Software Deployment and (Re)Configuration (DECOR'2004)*, pages 187–192, Grenoble, France, October 2004.
7. S. Lacour, C. Pérez, and T. Priol. Description and packaging of MPI applications for automatic deployment on computational grids. Research Report RR-5582, INRIA, IRISA, Rennes, France, May 2005.
8. S. Lacour, C. Pérez, and T. Priol. Generic application description model: Toward automatic deployment of applications on computational grids. In *6th IEEE/ACM Int. Workshop on Grid Computing (Grid2005)*. Springer, November 2005.
9. C. Pérez, T. Priol, and A. Ribes. A parallel CORBA component model for numerical code coupling. *The Int. Journal of High Performance Computing Applications*, 17(4):417–429, 2003.
10. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, Dec. 2002.

Towards Automatic Creation of Web Services for Grid Component Composition

Jan Dünnweber and Sergei Gorlatch
Dept. of Computer Science, University of Münster, Germany

Françoise Baude, Virginie Legrand and Nikos Parlavantzas[1]
OASIS team, INRIA/CNRS I3S/UNSA, France

Abstract. A promising approach to the development of grid software consists in combining high-level components, which simplify application programming, with Web services, which enable interoperability among distributed components. In this paper, we consider the combination of HOCs (Higher Order Components) and the ProActive/Fractal component model. Since instantiating skeletons requires transferring executable code, the following problem arises: how does one support code-carrying parameters in component interfaces exposed as web services? The paper shows how this problem has been solved by combining the ProActive/Fractal mechanism for automatic web service exposition with the HOC mechanism for code mobility.

1 Introduction

The complexity of developing grid applications is currently attracting much research attention. A promising approach for simplifying development and enhancing application quality is skeleton-based development [8]. This approach is based on the observation that many applications share a common set of recurring patterns such as divide and conquer, farm, and pipeline. The idea is then to capture such patterns as generic software constructs that can be customised by developers to produce multiple applications.
A recently proposed, skeleton-based approach is given by *Higher Order Components* (HOCs) [10] approach, which also exploits component technology and web services. HOCs are components for grid applications, customised with application-specific code, and exposed as web services. Since HOCs and the customizing code reside at different locations, the HOC approach includes support for code mobility. HOCs simplify application development because they isolate application developers from the details of building HOCs. The HOC approach can meet the requirements of providing a component architecture for grid programming with respect to abstraction and interoperability for two reasons: (1) the skeletal programming model offered by HOCs imposes a clear separation of concerns in terms of high-level services expecting from their users the provision of application-level code only, and (2) any HOC offers a publicly available interface in form of a Web service making it accessible to remote systems without introducing any requirements on them, e. g. , regarding the use of a particular middleware

[1] This work was carried out for the CoreGRID IST project n°004265, funded by the European Commission.

technology or programming language.
Building new grid applications using HOCs is simple as long as they require only HOCs that are readily available: In this case only some new parameter code must be specified. However, once an application adheres to a processing pattern that is not covered by the available HOCs, a new HOC has to be built. Building new HOCs currently requires starting from scratch and using low-level Grid middleware, which is tedious and error prone. We believe that combining another high-level Grid programming environment, such as GAT [6] or ProActive [3] can greatly reduce the complexity of developing and deploying HOC components. This complexity can be reduced further by providing support for composing HOCs out of other HOCs (e. g. , in a nested manner) or other reusable functionality. For this reason, we are investigating the uniform use of the ProActive/Fractal [7] component model for implementing HOCs as assemblies of smaller-grained components, and for integrating HOCs with other HOCs and client software.
Since HOCs need to be parameterised with code, the implementation of a HOCs as a ProActive/Fractal component poses the following technical problem: how can one pass code-carrying arguments to component interfaces exposed as web services? This paper describes how this problem is addressed by combining HOC's code mobility mechanism with ProActive/Fractal's mechanism for automatic web service exposition.
The rest of this paper is structured as follows. First, section 2 describes the HOC approach, focusing on the code mobility mechanism. Section 3 then discusses how HOCs can be implemented in terms of ProActive/Fractal components. Section 4 presents the solution to the problem of supporting code-carrying parameters, and section 5 concludes the paper.

2 Higher-Order Components (HOCs)

Higher-Order Components [10] [9] (or HOCs) have been defined with the aim to provide an efficient, parallel grid implementation of skeletons. HOCs are customised by plugging application-specific code to appropriate places in the skeleton, and they expose web service enabled interfaces for customisation and triggering computations. Specifically, a HOC client first invokes the customisation services exposed by the HOC. The goal is to set the behaviours that are left open in the skeleton, e.g., the behaviours of masters and workers in a farm HOC. Next, the client invokes services on the configured HOC to initiate computation and retrieve the results. For interoperability purposes between the client and the instances of HOCs, all those services are exposed as web services.
Let us demonstrate this feature using the example of an HOC implementing the Farm skeleton, with a master and an arbitrary number of workers.

This HOC implements the dispatching of data emitted from the master via scattering, i. e. , each worker is sent an equally sized subset of the input. The HOC implementation is partial since it does neither include the required code to split an input array into subsets, nor the code to process one single subset. These application-specific codes must be specified by the client.

More precisely, the client must provide (in a registry) code units that correspond to the following interfaces:

```
public interface Worker {
    public double[] compute(double[] input);
}
public interface Master {
    public double[][] split (double[] input, int numWorkers);
    public double[] join(double[][] input);
}
```

The client creates, customises, and triggers a farm HOC in the following way:

```
farmHOC =farmFactory.createHOC();
farmHOC.setMaster("masterID"); // web service invocation in Java
farmHOC.setWorker("workerID");
String[] targetHosts = {"masterH", "workerH1", ...};
farmHOC.configureGrid(targetHosts); // deployment of the farmHOC on the Grid
farmHOC.compute(input);
```

Note that we have chosen `double`-arrays as parameter types for the code units because it is actually the most general type possible, as the type must have a corresponding representation in the WSDL types element, which is an XML-Schema. The `xsd:any` type would not help at all, since the Globus Toolkit can only convert it to plain Java Object and forbids all derived types, as they cannot be serialized/deserialized in a transmission when the defining class is not present on both, the sender and the receiver side. A plain `java.lang.Object` is not suitable to transmit any significant information and alternatives like Java Beans (i. e. , classes composed of attributes and corresponding accessors only) result in fixed parameter types and also require a much more extensive marshaling process than the primitive `double` type.

An important feature of the HOC architecture is the ’code mobility’ mechanism, which supports shipping code units from a registry where clients have put them previously to HOCs for customising the skeleton. In our example, the code shipping mechanism is used when a client invokes `setMaster("masterID")` on the farm HOC. The ”masterID” identifies the code unit that has been previously placed in the registry.

In the particular case of code units being Java bytecode, the code mobility mechanism employs a remote class loader for HOCs that replaces the Java default class loader. The introduced loader connects to a specific service (named the *Code Service* in the HOC Service Architecture) instead of searching class files locally when new classes are loaded. After the bytecode for a particular class is retrieved from a remote registry, the class is instantiated by the introduced loader using the Java reflection mechanism. Overall, the code mobility mechanism provides a sort of mapping of code parameters implementing the Java interfaces required for a given type of HOC to XML-schema definitions, as they are used in WSDL descriptions. This mapping is indirect as it relies on the usage of string identifiers for code parameters (which can be obviously be expressed in the corresponding WSDL). Also, the current implementation of the HOC architecture does not enable the automatic generation of WSDL descriptions associated

with the services of HOCs to be published. It is the duty of the programmer of a (new type of) HOC to provide the code for the WSDL generation.

3 Higher-Order Components (HOCs) built upon ProActive/Fractal

ProActive/Fractal components are runtime entities that communicate using interfaces connected through bindings. An important feature of the model is its support for hierarchical composition, i. e. , components can be connected and nested into each other up to arbitrary levels of abstraction. Component configurations can be specified flexibly using an architecture description language (ADL) and mapped declaratively to arbitrary network topologies using deployment descriptors. Moreover, components can be associated with an extensible set of controllers, which enable inspecting and reconfiguring their internal features. Importantly, the model also supports automatically exposing component interfaces as Web services, thus enabling interoperation across different programming languages. The component model is expected to simplify developing and modifying HOCs because it presents a high abstraction level to developers and supports easily changing configurations and deployment properties without code modifications.
Using ProActive/Fractal, a HOC will be formed as a composite that contains components customizable with externally-provided behaviour. Consider, again, the Farm-HOC from the previous section, and see how it could be seen as a Fractal component. This would be a composite component containing a primitive component called master connected to an arbitrary number of other primitives called workers (fig. 1). The master and the workers can reside either on a single machine or they can be distributed over multiple nodes of the grid, depending on the deployment configuration. For even more efficiency, the master could dispatch data to workers using the built-in scattering (group communication) mechanism provided by the component model.
The master and worker components are customisable with external behaviour (depicted with the black cycles) through the following interface exposed on the composite:

```
public interface Customisation {
   public void setMaster(Master m);
   public void setWorker(Worker w);
}
```

Following the HOC architecture, the `Customisation` interface must be exposed as a web service. However, this requires that one can pass a code-carrying, behavioural argument (e.g., a `Master` implementation) to a web service, which is currently unsupported in ProActive/Fractal. This technical problem is solved using the code mobility mechanism of the HOC architecture combined with the existing ProActive mechanisms for automatic web service exposition.

4 Accessing HOC components via ProActive web services

This section first describes the existing ProActive mechanism for automatically exposing components as web services, and then it explains how the mechanism has been extended to solve the technical problem identified earlier.

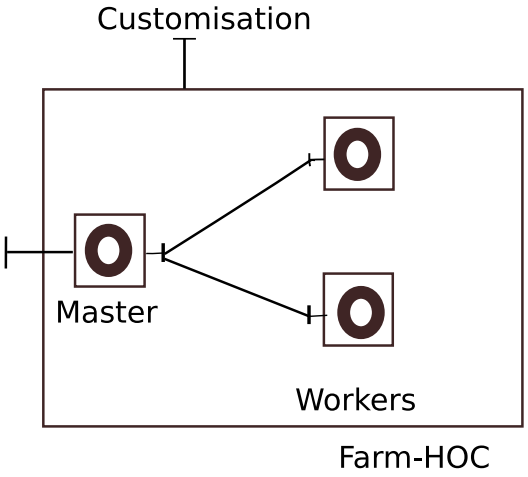


Fig. 1. The Farm-HOC shown using the Fractal symbols

ProActive allows any web service enabled client to invoke services on component interfaces. The implementation uses the Axis [2] library to generate the appropriate WSDL description and the Apache SOAP [1] engine to deploy and route the service invocation through a custom ProActive *provider*. Exposing components as web services is performed through simply using the ProActive static method `exposeComponentAsWebService`, which generates the service description and makes it available on the web server. This mechanism supports all defined types in the SOAP specification; Java primitive types are supported, but not complex types. When consumers need to perform a call on a service, they get the description and just perform the call according to the WSDL contract (fig. 2, step 1).

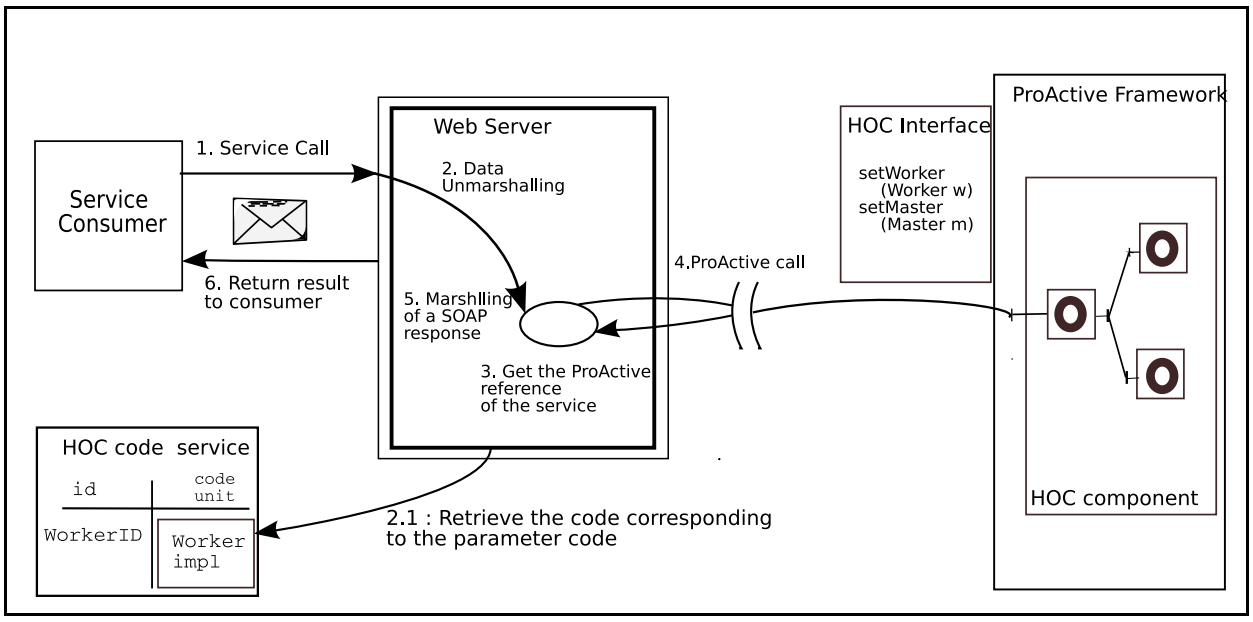


Fig. 2. ProActive web services mechanism with HOC remote class loading

This architecture removes the need to worry about processing SOAP messages: when a call reaches the provider, the engine has already unmarshalled the SOAP message and knows which method to call on which object (fig. 2, step 2). The provider needs only to implement the logic required to call the corresponding public methods. Specifically, the provider gets a remote reference on the targeted interface (fig. 2, step 3), it performs a standard ProActive call from the web server side to the remote ProActive runtime side using the reference (fig. 2, step 4), and it returns the result to the SOAP engine. The engine then marshalls a new SOAP message and sends it back to the service consumer (fig. 2, steps 5 and 6).
As mentioned earlier, the problem with the existing mechanism is that passing arguments of complex type is unsupported, which is unsatisfactory as it is required, in particular, to pass code-carrying arguments for customizing HOCs. We do aim to solve the problem in the general case, but we have addressed it for the specific case of HOCs implemented as Fractal/ProActive components. For this purpose, the mechanism of exposing components as web services has been extended to use the HOC remote class loading.
This extension affects WSDL generation at the deployment time, but also the routing step at the calling time:

- First, we generate a WSDL description that maps behavioural parameters to identifiers used to denote code units in the HOC code service. Moreover, the description identifies explicitly the parameters that correspond to behavioural parameters, using XML schema annotations.
- Second, we extend the ProActive provider to retrieve the right code unit according to the *identifier* the client sent (fig. 2, step 2.1).This *extended provider* integrates the remote code loading mechanism explained in section 2. Once the code unit has been retrieved, the provider performs the call on the component interface and sends back any result to the client.

Since the transfer of code parameters between clients and HOCs is handled using SOAP our mixed implementation of Fractal and the HOC class loading mechanism introduces a slight performance overhead during the initialisation phase of an application, as compared to using only ProActive for handling all data transfer. For the farm HOC, we measured, e. g. , that installing some Worker code of 5KB length takes about 100ms at average. This step has to be repeated for each Worker host. So, if, e. g. , 10 worker hosts run this 5KB code parameter, approximately 1 additional second installation time will be needed. Marginal performance reductions like this can of course be disregarded relative to the typical runtimes of grid applications. It should also be noted that this time is spend only once during a non-recurring setup step, when a component is accessed for the first time.

5 Conclusion and Perspectives

This paper has described a solution to supporting code-carrying parameters in component interfaces, offering transparency to developers at the code receiving side. A natural

direction for future work is to provide tools for interpreting WSDL descriptions containing such parameters in order to provide transparency also at the code sending side. Further work would also be to devise a general solution to supporting any type, even a complex Java type, when publishing ProActive/Fractal components as web services. Indeed, this paper only presents a first step in this direction, as it exposes a solution that only applies to some specific parameter types, i.e. those representing behaviours. More concretely, the general case would call for a solution where the generation of the extended ProActive provider would be totally automated. The solution presented here is specific in the sense that the extended provider has been generated specifically for the case of HOC.

For addressing the general case we should be aware of some relevant previous or current works: (1) the *valuetype* construct in CORBA, which supports passing objects by value (both state and behaviour) to remote applications [4], (2) possible – not yet standard – extensions of WSDL for passing arguments as complex types using specific SOAP attachments, and (3) standard facilities for XML data binding, such as the Java Architecture for XML Binding 2.0 JAXB [5]. Whatever the solution we would use for passing parameters of arbitrary types, it calls for a generic and automatic mechanism based on reflection techniques and dynamic code generation.

The paper has also discussed how HOCs can be implemented as composite components in the ProActive/Fractal component model. Our work can thus be considered as an interesting joint effort to devise grid enabled skeletons based on a fully-fledged component oriented model, effectively using the dynamic (re)configuration capabilities, and the ability to master complex codes through hierarchical composition. Doing this, we foresee that a skeleton would be configured by passing it fully-fledged software components as its internal entities. The configuration options could be made even wider than in the current HOC model, by adding specific controllers on the composite component representing a whole skeleton, that could recursively also affect the included components.

References

1. The apache soap web site. http://ws.apache.org/soap/.
2. The axis web site. http://ws.apache.org/axis/.
3. The proactive web site. http://www-sop.inria.fr/oasis/ProActive/.
4. *CORBA/IIOP v3.0.3*. Object Management Group, 2004. OMG Document formal/2004-03-01.
5. *The Java Architecture for XML binding 2.0, early draft v0.4*. Sun Microsystems, 2004.
6. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. v. Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The grid application toolkit: Towards generic and easy application programming interfaces for the grid. In *Proceedings of the IEEE, vol. 93, no. 3*, pages 534 – 550, 2005.
7. F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, 2003.
8. M. I. Cole. *Algorithmic skeletons: a structured approach to the management of parallel computation*. MIT Press & Pitman, 1989.

9. J. Dünnweber and S. Gorlatch. HOC-SA: A grid Service Architecture for Higher-Order Components. In *International Conference on Services Computing (SCC04), Shanghai, China*, pages 288–294, Washington, USA, 2004. IEEE computer.org.
10. S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.

Using Code Parameters for Component Adaptations

Jan Dünnweber, Sergei Gorlatch[1], Sonia Campa, Marco Danelutto[2], and Marco Aldinucci[3]

[1] Dept. of Computer Science – University of Münster – Germany
[2] Dept. of Computer Science – University of Pisa – Italy
[3] Inst. of Information Science and Technologies – CNR, Pisa, Italy

Abstract. Adaptation means that the behavior of a software component is adjusted to application- or platform-specific requirements: new components required in a particular application do not need to be developed from scratch when available components can be adapted accordingly. Instead of introducing a new adaptation syntax (as it is done, e. g. , in AOP), we describe adaptations in the context of Java-based Higher-Order Components (HOCs).
HOCs incorporate a code parameter plugin mechanism enabling adaptations on the grid. Our approach is illustrated using a case study of sequence alignment. We show how a HOC with the required provisions for data dependencies in this application can be generated by adapting a farm component, which is ”embarrassingly parallel”, i. e. , free of data dependencies. This way, we could reuse the efficient farm implementation from the Lithium library, although our case study exhibits the wavefront pattern of parallelism which is different from the farm.

1 Introduction

This paper addresses grid application programming using a component framework, where applications are built by *selecting*, *customizing* and *combining* components. Selecting means choosing appropriate components from the framework repository, which may contain several ready-made implementations of commonly used parallel computing schemata, e. g. , algorithmic skeletons (farm, divide-and-conquer, etc.) [4]. By customization, we mean specifying application-specific operations to be executed within the processing schema of a component, e. g. , parallel farming of application-specific tasks. Combining various parallel components together can be done, e. g. , via Web services.

As our main contribution, we introduce *adaptations* of software components, which extends the traditional notion of *customization*: while customization applies a component’s computing schema in a particular context, adaptation modifies the very schema of a component, with the purpose of incorporating new capabilities. Our thrust to use more flexible, adaptable components is motivated by the fact that a fixed component framework is hardly able to cover all possible processing schemata. The sequential and parallel behavior of adaptable components can be altered, thus allowing to apply them in use cases for which they have not been originally designed. We demonstrate that both, traditional customization and adaptation of components can be realized in a grid-aware manner using code parameters that can be shipped over the network of a grid.

As a case study, we take a component that was originally designed for dependency-free *task farming*. By means of an additional code parameter, we adapt this component for the parallel processing exhibiting data dependencies with a *wavefront* structure.

In Section 2, we explain our *Higher-Order Components* (HOCs) and how they can be made adaptable. Section 3 describes our application case study used throughout the paper: the alignment of sequence pairs, which is a wavefront-type, time-critical problem in computational molecular biology [7]. In Section 4, we show how the HOC-framework enables the use of mobile code, as it is required to apply a component adaptation in the grid context, and present our grid-like testbed, highlighting the settings relevant for the system's adaptivity. Section 5 shows our first experimental results for the alignment problem in different, grid-like infrastructures. Section 6 summarizes the contributions of this paper in the context of related work.

2 Higher-Order Components (HOCs) and the Farm pattern

Higher-Order Components (HOCs) [6] are called so because they can be parameterized not only with data but also with code. We illustrate the HOC concept using a particular component, the Farm-HOC, which will be our example throughout the paper.

The farm pattern is a popular pattern of "embarrassing parallelism", without dependencies between tasks. There may be different implementations of the farm, depending on the target computer platform; all these implementations have, however, in common that the input data are partitioned using a code unit called the `Master` and the tasks on the data parts are processed in parallel using a code unit called the `Worker`. The component expressing the farm schema, the Farm-HOC, has therefore two so-called *customization code parameters*, the `Master`-parameter and the `Worker`-parameter, defining the corresponding code units in the farm implementation.

These two parameters specify how the general farm schema should be applied in a particular situation. The `Master` parameter must contain a `split`-method for partitioning the input data and a corresponding `join` method for recombining it, while the `Worker` parameter must contain a `compute`-method for task processing. To use the Farm-HOC in our Java-based, grid-aware component framework, the programmer must provide implementations of the following two interfaces:

```
1: public interface Master<E>  {
2:  public E[][] split(E[] input, int grain);
3:  public E[] join(E[][] results);   }
4: public interface Worker<E>  {
5:  public E[] compute(E[] input);  }
```

The `Master` (line 1–3) determines how an input array of some type `E` is split into independent subsets and the `Worker` (line 4–5) describes how a single subset is processed as a task in the farm. While the `Worker`-parameter differs in most applications, a specific implementation of the `Master` only has to be provided, if the input of a particular application should not be subdivided regularly, but it requires a special decomposition algorithm, e. g. , for preserving certain data correlations. Thus, in most applications, the user will only specify the `Worker` and pick a default `Master` implementation from our framework.

3 Case Study: Sequence Alignment

We illustrate the motivation for adaptation and its use by the following application case study.

One of the fundamental algorithms in bioinformatics is the computation of *distances* between DNA sequences, i. e., finding the minimum number of insertion, deletion or substitution operations needed to transform one sequence into another. Sequences are encoded using the alphabet $\{A,C,G,T\}$, where each letter stands for one of the nucleotide types [3].

The distance, which is the total number of the required transformations, quantifies the similarity of sequences [8] and is often called *global alignment* [12]. Mathematically, global alignment can be expressed using a so-called *similarity matrix S*, whose elements $s_{i,j}$ are defined as follows:

$$s_{i,j} := max\left(s_{i,j-1}+plt, s_{i-1,j-1}+\delta(i,j), s_{i-1,j}+plt\right) \tag{1}$$

where

$$\delta(i,j) := \begin{cases} +1 \text{ , if } \varepsilon_1(i) = \varepsilon_2(j) \\ -1 \text{ , otherwise} \end{cases} \tag{2}$$

In Definition2 $\varepsilon_k(b)$ denotes the b-th element of sequence k, and plt is a constant that weighs the costs for inserting a space into one of the sequences (typically, $plt = -2$, the "double price" of a mismatch).

The wavefront pattern of parallel computation is not specific only to the sequence alignment problem, but is used also in other popular applications: searching in graphs represented via their adjacency matrices, system solvers, character stream conversion problems, motion planning algorithms in robotics etc. Therefore, programmers would benefit if a standard component, such as a HOC, would capture the wavefront pattern.

Our approach is to take the Farm-HOC, as introduced in Section 2, adapt it to the required wavefront structure of parallelism and then customize it to the sequence alignment application.

Fig. 1 schematically shows this two-step procedure. First, the workspace, holding the partitioned tasks for farming, is sorted according to the wavefront pattern, whereby a new processing order is fixed, which is optimal with respect to the degree of parallelism. Then, the alignment definitions (1) and (2) are employed, determining how to process single input data elements. Finally, this adapted component can be used for processing the sequence alignment application.

4 Adaptation with Globus & WSRF

Let us take a closer look at the currently most modern version of the Globus middleware and the enclosed implementation of the *Web Services Resource Framework* (WSRF) [9], before we present our extensions of this middleware for simplifying application development and for enabling component adaptations. WSRF allows to set up stateful resources and connect them to Web services. Such resources can represent application state data and thereby make Web services and their XML-based communication protocol (SOAP) more suitable for grid computing: while usual Web services

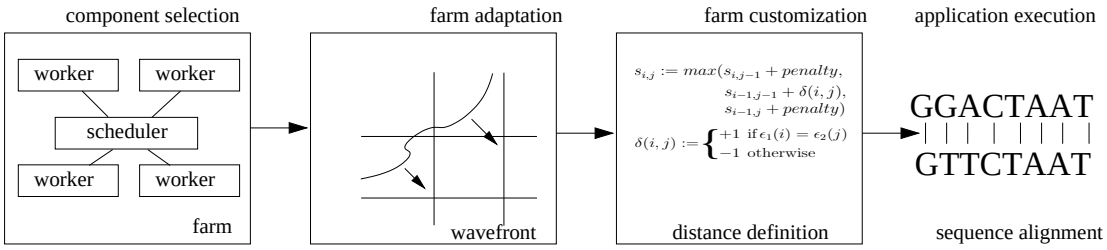


Fig. 1. Two-step process: adaptation and customization

offer only self-contained operations, which are decoupled from each other and from the caller, Web services hosted with Globus include the notion of a context; i. e. , multiple operations can affect the same data and changes within this data can trigger callbacks to the service consumer avoiding blocking invocations.

While making Web services more eligible for performance-critical applications, Globus is still too low-level to be used directly by application programmers: it requires the programmer to manually write multiple XML-configuration files and to place them properly within the grid servers' installation directories. These files must explicitly declare all resources, the services used to connect to them, their interfaces and the corresponding bindings to the employed protocol, in order to make Globus applications accessible in a platform- and programming language-independent manner.

4.1 Enabling Mobile Code

Programming with adaptable and customizable components requires, besides the exchange of data, the exchange of *mobile code* across network boundaries. Therefore, we provide a special class-loading mechanism allowing class definitions to be exchanged among distributed servers. Interconnections between servers, which execute HOCs, and clients are established according to the WSRF standard.

Users of the HOC-framework are completely freed from the complicated WSRF-setup described above, as all the required files, which are specific for each HOC but independent from applications, are provided for all the available HOCs in advance.

In the following, we illustrate the two-step process of adaptation and customization shown in Fig. 1. For the sake of explanation, we start with the second step (HOC customization), and then consider the farm adaptation.

4.2 Customizing the Farm-HOC for Sequence Alignment

The farm pattern is only one of many possible patterns of parallelism, arguably one of the simplest, which is available in many parallel component frameworks. When an application requires another component, which is not provided by the employed framework, there are two possibilities: either to code the required component anew or to try and derive it from another available component. The former possibility is more direct, but it has to be done repeatedly for each new application. The latter possibility, which we call adaptation, provides more flexibility and potential for reuse of components.

However, it requires from the employed framework to have a special mechanism for enabling such adaptations.

Our framework includes a straightforward `Master` implementation for matrices, which partitions matrices into equally sized submatrices and recombines the submatrices after they have been processed. So, in the case of a matrix application, we do not need to write our own `Master` code parameter for partitioning the input data, but we can fetch the framework procedure from the code service by passing its ID (`matrixSplit`) to the Farm-HOC. The only code parameter we must write anew for computing the similarity matrix in our sequence alignment application is the `Worker` code. In our case study this parameter implements, instead of the general `Worker`-interface, the alternative `Binder`-interface, which describes, specifically for matrix applications, how an element is computed depending on its indices:

```
1: public interface Binder<E>  {
2:  public E bind(int i, int j);  }
```

Before the HOC computes the matrix elements, it assigns an empty workspace matrix to the code parameter; i. e. , a `matrix` reference is passed to the parameter object and, thus, made available to the customizing parameter code for accessing the matrix elements.

Our code parameter implementation for calculating matrix elements, accordingly to definition (1) from section 3, reads as follows:

```
1: new BinderParameter<Integer>( ) {
2:  public Integer bind(int i, int j)  {
3:  return max( matrix.get(i, j - 1) + penalty,
4:   matrix.get(i - 1, j - 1) + delta(i, j),
5:   matrix.get(i - 1, j) + penalty );  }  }
```

The helper method `delta`, used in line 4 of the above code, implements definition (2). The special `Matrix`-type used for representing the distributed matrix data being split up among the workers by the HOC is provided by our framework and it facilitates full location transparency, i. e. , it allows to use the same interface for accessing remote elements and local elements. Actually `Matrix` is an abstract class and our framework includes two concrete implementations: `LocalMatrix` and `RemoteMatrix`. These classes allow to access elements in neighboring submatrices using overhang-indices (including negatives), which further simplifies the programming of distributed matrix algorithms. Obviously, these framework-specific utilities are quite helpful in the presented case study. Anyway, they are not necessary neither for customizing nor for adapting software components on the grid. Therefore, the implementation of these auxiliary classes is beyond the scope of this paper.

Farming the tasks described by the above `BinderParameter`, i. e. , the matrix element computations, does not allow data dependencies between the elements. Therefore any farm implementation, including the one available in the Lithium library used in our case, would compute the alignment result as a single task, without parallelization, which is unsatisfactory and will be addressed by means of adaptation.

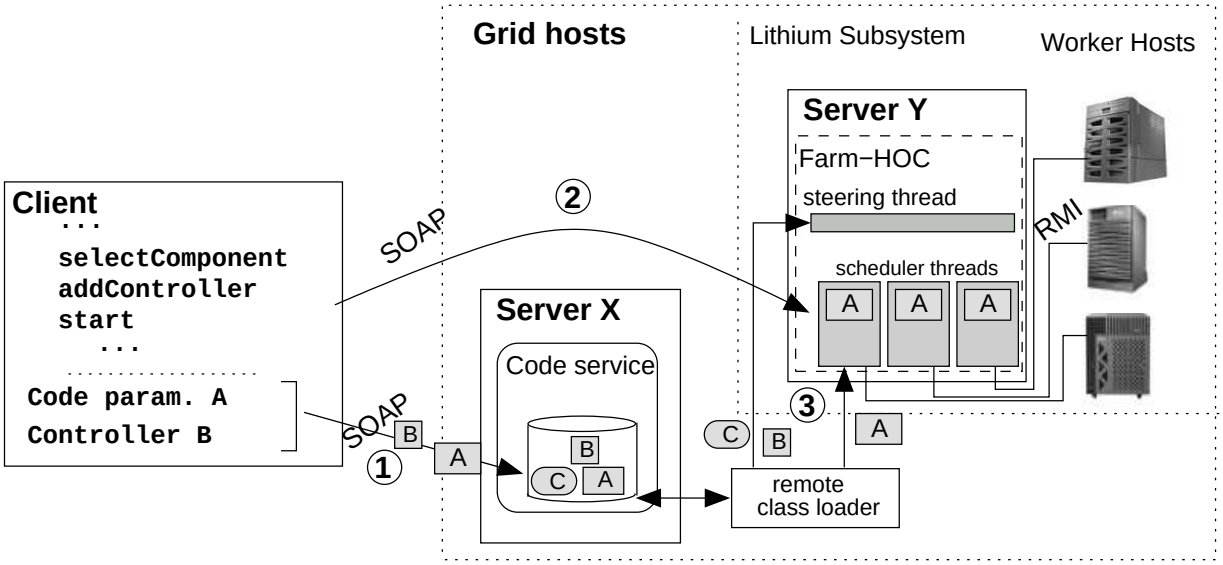


Fig. 2. Running the adapted component on the grid

4.3 Adapting the Farm-HOC to the Wavefront Pattern

In this section, we adapt the Farm-HOC to the wavefront pattern, so that it can be used for our example application. Like the farm customization described in the preceding section, the adaptation of the farm's parallel behavior is handled by means of code parameters, which are handled using our remote class loader and the code service providing a grid-aware code transfer mechanism as introduced above.

For the parallel processing of submatrices, the adapted component must, initially, fix the "wavefront order" for processing individual tasks, which is done by sorting the partitions of the workspace matrix arranged by the `matrixSplit-Master` from the HOC-framework, such that independent submatrices are grouped in one wavefront. We compute this sorted partitioning, while iterating over the matrix-antidiagonals as a preliminary step of the adapted farm, similar to the loop-skewing algorithm described in [11].

The central role in our adaptation approach is played by the special *steering thread* that is installed by the user and runs the wavefront-sorting procedure in its initialization method. In this method, we also initialize the border row and column of the similarity matrix S, in our implementation.

4.4 Configuring the Runtime Environment

The client starts the configuring the runtime environment by uploading two code parameters, A and B, to the code service (step ① in Fig. 2). Parameter A is the farm worker parameter applying the `bind`-method from section 4.2; parameter B is the steering thread; i. e. , it defines the adaptation of the farm for wavefront processing. Parameter C , which represents the `Master`, is the only parameter not uploaded

by the client, but readily provided by the `matrixSplit`-implementation in our framework. In the final configuration step ③, Server Y retrieves the code parameters [A], [B] and (C) from the code service and installs them using the remote class loader.

Our Farm-HOC, which is now adapted to wavefront computations and customized for sequence alignment, then handles the whole distributed computation process on behalf of the client, which receives a notification once the process is finished.

5 Experimental Results

To investigate the scalability of our implementation over several servers, we ran it using two Pentium III servers under Linux at 800MHz. In the left plots in Fig. 3, we investigated the scalability using two multiprocessor servers: the U880 plus a second SunFire 6800 with 24 1350 Mhz UltraSPARC-IV processors. As can be seen, the performance of our applications is significantly increased for the 32 processor configuration, since the SMP-machine-interconnection does not require the transmission of all tasks over the network, for dispatching them to multiple processors. Curves for the standard farm are not shown in these diagrams, since they lie far above the shown curves and coincide for 8 and 32 processors, which only proves again that this version does not allow for parallelism within the processing of a single sequence pair.

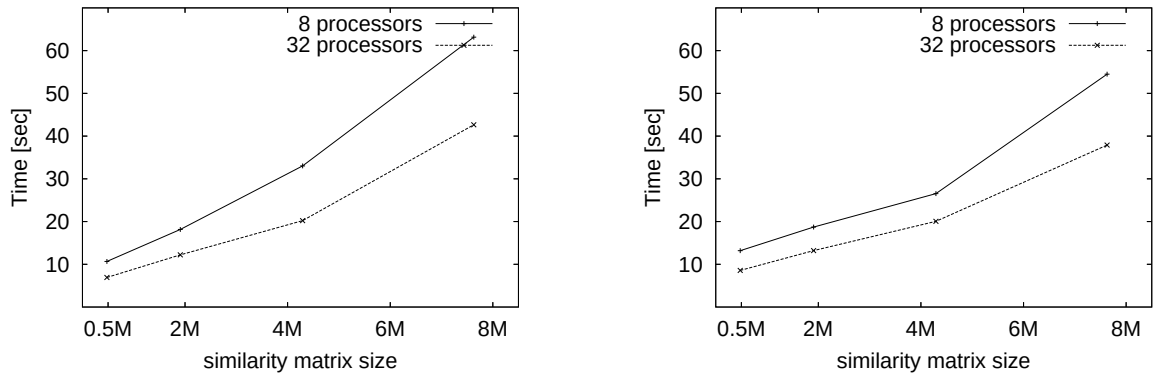


Fig. 3. Experimental results on grid-like testbeds. left: multiple multiprocessor servers; right: same input, zipped transmission

The right plots in Fig. 3 show the effect of another interesting modification: When we compress the submatrices using the Java `util.zip Deflater`-class, before we transmit them over the network, the curves do not grow so fast, since the compression procedure slows down the process, for small-sized input, but the absolute times for larger matrices are improved.

6 Conclusion and Related Work

As its main contribution, this paper introduced, implemented and experimentally investigated a novel method for the adaptation of parallel programming components, in order

to optimize their behavior for grid applications. We have shown that the code parameter mechanism provided by our Higher-Order Components (HOCs) allows for building grid-aware applications via adaptation, which can free the programmer from developing and deploying new components in many use cases. To the best of our knowledge, adaptations of components have so far not been considered, neither in the general component model [10] nor in the skeleton approach to parallel programming [4]. Adaptation extends the previous notion of component customization, which was restricted to only specifying the computation part of a component.

Our farm implementation was taken from the Java-based system Lithium [5]. In [6], we described an alternative Farm-HOC implementation, in which not only a Web service was used to connect to the HOC, but also the communication within the farm itself was realized using Web services deployed into a Globus container. Generally any middleware, e. g. , CORBA or MPICH [2], can be used for providing HOC implementations, as long as the format used for representing mobile code is supported by the chosen technology. We use the Java-based Lithium system in this paper, because our alignment application is also written in Java, and because it requires frequent communication between the scheduler and the workers, which can be handled more efficiently via RMI as done in Lithium than via SOAP. Framework implementations, like Lithium, should be distinguished from abstract component models, like CCA or Fractal where adaptations of components are principally possible, but it is not specified how to apply them. The messaging model we used for stopping the farm activity, whenever data dependencies prevented continuation, does not require anything more then a possibility for broadcasting messages among processes. It can therefore be realised in the same way, in a CCA implementation like CCaffeine or in Julia (The reference implementation of Fractal).

Possible alternatives to the described adaptation of the Farm-HOC for wavefront algorithms include replacing the Lithium farm scheduler by another one operating in a wavefront manner or adding a completely new wavefront HOC to our component framework. The first alternative would be valid only for the Lithium system and, moreover, we would hard-wire the wavefront behavior into the farm. The second alternative involves more overhead for the programmer than the adaptation of an existing component.

The use of the wavefront schema for parallel sequence alignment has been analyzed before in [1], where it is classified as a design pattern. While in the CO_2P_3S system the wavefront behavior is a fixed part of the pattern implementation, in our approach, it is only one of many possible adaptations that can be applied to a HOC. Since our wavefront steering thread can also be plugged into the scheduler of the Lithium library without uploading it remotely, our solution can be viewed as a novel way of introducing a new skeleton to a skeleton-library, without changing its implementation.

Acknowledgment

This research was conducted within the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

1. J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan. Generating parallel programs from the wavefront design pattern. In *7th Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE Computer Society Press, 2002.
2. Argonne National Laboratory. The Message Passing Interface (MPI). http://www-unix.mcs.anl.gov/mpi.
3. C.-I. Branden, J. Tooze, and C. Branden. *Introduction to Protein Structure*. Garland Science, 1991.
4. M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. Pitman, 1989.
5. M. Danelutto and P. Teti. Lithium: A structured parallel programming enviroment in Java. In *Proceedings of Computational Science - ICCS*, number 2330 in Lecture Notes in Computer Science, pages 844–853. Springer-Verlag, Apr. 2002.
6. S. Gorlatch and J. Dünnweber. From grid middleware to grid applications: Bridging the gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2005.
7. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1999.
8. V. I. Levenshtein. Binary codes capable of correcting insertions and reversals. In *Soviet Physics Dokl. Volume 10*, pages 707–710, 1966.
9. OASIS Technical Committee. WSRF: The Web Service Resource Framework, http://www.oasis-open.org/committees/wsrf.
10. C. Szyperski. *Component software: Beyond object-oriented programming*. Addison Wesley, 1998.
11. M. Wolfe. Loop skewing: the wavefront method revisited. In *Journal of Parallel Programming, Volume 15*, pages 279–293, 1986.
12. X.Huang, R. Hardison, and W.Miller. A space-efficient algorithm for local similarities. In *Computer Applications in the Biosciences*, volume 6(4), pages 373–381. Oxford University Press, 1990.

Towards the Automatic Mapping of ASSIST Applications for the Grid

Marco Aldinucci[1] and Anne Benoit[2]

[1] Inst. of Information Science and Technologies
National Research Council (ISTI–CNR)
Via Moruzzi 1, Pisa I-56100, Italy
aldinuc@di.unipi.it
[2] LIP, Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Anne.Benoit@ens-lyon.fr

Abstract. One of the most promising technical innovations in present-day computing is the invention of grid technologies which harness the computational power of widely distributed collections of computers. However, the programming and optimisation burden of a low level approach to grid computing is clearly unacceptable for large scale, complex applications. The development of grid applications can be simplified by using high-level programming environments. In the present work, we address the problem of the mapping of a high-level grid application onto the computational resources. In order to optimise the mapping of the application, we propose to automatically generate performance models from the application using the process algebra PEPA. We target in this work applications written with the high-level environment ASSIST, since the use of such a structured environment allows us to automate the study of the application more effectively.

Key words: high-level parallel programming, grid, ASSIST, PEPA, automatic model generation, skeletons.

1 Introduction

A grid system is a geographically distributed collection of possibly parallel, interconnected processing elements, which all run some form of common grid middleware (e.g. Globus services) [15]. The key idea behind grid-aware applications is to make use of the aggregate power of distributed resources, thus benefiting from a computing power that falls far beyond the current availability threshold in a single site. However, developing programs able to exploit this potential is highly programming intensive. Programmers must design concurrent programs that can execute on large-scale platforms that cannot be assumed to be homogeneous, secure, reliable or centrally managed. They must then implement these programs correctly and efficiently. As a result, in order to build efficient

grid-aware applications, programmers have to address the classical problems of parallel computing as well as grid-specific ones:

1. Programming: code all the program details, take care about concurrency exploitation, among the others: concurrent activities set up, mapping/scheduling, communication/synchronisation handling and data allocation.

2. Mapping & Deploying: deploy application processes according to a suitable mapping onto grid platforms. These may be highly heterogeneous in architecture and performance. Moreover, they are organised in a cluster-of-clusters fashion, thus exhibiting different connectivity properties among all pairs of platforms.

3. Dynamic environment: manage resource unreliability and dynamic availability, network topology, latency and bandwidth unsteadiness.

Hence, the number and quality of problems to be resolved in order to draw a given QoS (in term of performance, robustness, etc.) from grid-aware applications is quite large. The lesson learnt from parallel computing suggests that any low-level approach to grid programming is likely to raise the programmer's burden to an unacceptable level for any real world application.

Therefore, we envision a layered, high-level programming model for the grid, which is currently pursued by several research initiatives and programming environments, such as ASSIST [21], eSkel [10], GrADS [19], ProActive [7], Ibis [20], Higher Order Components [12, 13]. In such an environment, most of the grid specific efforts are moved from programmers to grid tools and run-time systems. Thus, the programmers have only the responsibility of organising the application specific code, while the programming tools (i.e. the compiling tools and/or the run-time systems) deal with the interaction with the grid, through collective protocols and services [14].

In such a scenario, the QoS and performance constraints of the application can either be specified at compile time or varying at run-time. In both cases, the run-time system should actively operate in order to fulfil QoS requirements of the application, since any static resource assignment may violate QoS constraints due to the very uneven performance of grid resources over time. As an example, ASSIST applications exploit an autonomic (self-optimisation) behaviour. They may be equipped with a QoS contract describing the degree of performance the application is required to provide. The ASSIST run-time environment tries to keep the QoS contract valid for the duration of the application run despite possible variations of platforms' performance at the level of grid fabric [6, 5]. The autonomic features of an ASSIST application rely heavily on run-time application monitoring, and thus they are not fully effective for application deployment since the application is not yet running. In order to deploy an application onto the grid, a suitable mapping of application processes onto grid platforms should be established, and this process is quite critical for application performance.

This problem can be addressed by defining a performance model of an ASSIST application in order to statically optimise the mapping of the application onto a heterogeneous environment, as shown in [1]. The model is generated from the source code of the application, before the initial mapping. It is expressed with the process algebra PEPA [17], designed for performance evaluation. The

use of a stochastic model allows us to take into account aspects of uncertainty which are inherent to grid computing, and to use classical techniques of resolution based on Markov chains to obtain performance results. This static analysis of the application is complementary with the autonomic reconfiguration of ASSIST applications, which works on a dynamic basis. In this work we concentrated on the static part to optimise the mapping, while the dynamic management is done at run-time. It is thus an orthogonal but complementary approach.

Structure of the paper. The next section introduces the ASSIST high-level programming environment and its run-time support. Section 3 introduces the Performance Evaluation Process Algebra PEPA, which can be used to model ASSIST applications. These performance models help to optimise the mapping of the application. We present our approach in Section 4, and give an overview of future working directions. Finally, concluding remarks are given in Section 5.

2 The ASSIST environment and its run-time support

ASSIST (A Software System based on Integrated Skeleton Technology) is a programming environment aimed at the development of distributed high-performance applications [21, 3]. ASSIST applications should be compiled in binary packages that can be deployed and run on grids, including those exhibiting heterogeneous platforms. Deployment and run is provided through standard middleware services (e.g. Globus) enriched with the ASSIST run-time support.

2.1 The ASSIST coordination language

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of modules, interconnected by typed streams of data. Each stream realises a one-way asynchronous channel between two sets of endpoint modules: sources and sinks. Data items injected from sources are broadcast to all sinks. All data items injected into a stream should match the stream type.

Modules can be either sequential or parallel. A sequential module wraps a sequential function. A parallel module *(parmod)* can be used to describe the parallel execution of a number of sequential functions that are activated and run as *Virtual Processes* (VPs) on items arriving from input streams. The VPs may synchronise with the others through barriers. The sequential functions can be programmed by using a standard sequential language (C, C++, Fortran). A *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel (e.g. farm) way and it may exploit a distributed shared state that survives the VPs lifespan. A module can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to instantiate VPs according to the input and distribution rules specified in the parmod. The VPs may send items or parts of items onto the output streams, and these are gathered according to the output rules. Details on the ASSIST coordination language can be found in [21, 3].

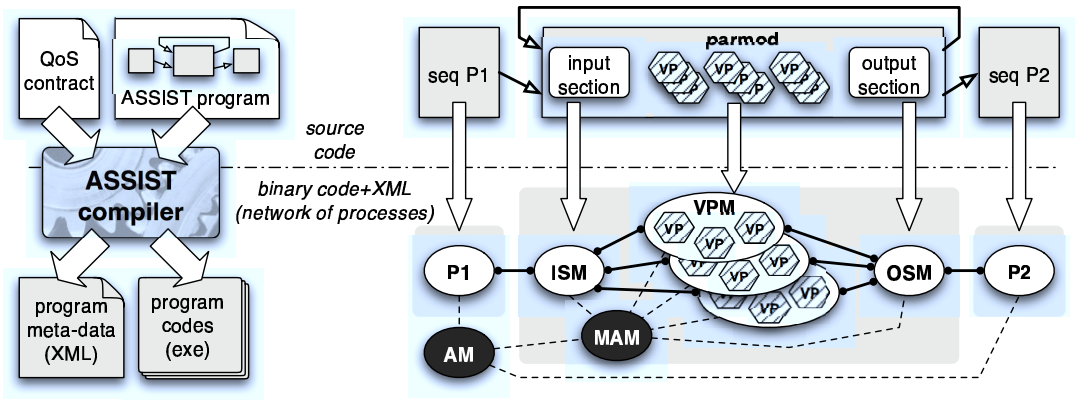


Fig. 1. An ASSIST application and a QoS contract are compiled in a set of executable codes and its meta-data [3]. This information is used to set up a processes network at launch time.

2.2 The ASSIST run-time support

The ASSIST compiler translates a graph of modules into a network of processes. As sketched in Fig. 1, sequential modules are translated into sequential processes, while parallel modules are translated into a parametric (w.r.t. the parallelism degree) network of processes: one *Input Section Manager* (ISM), one *Output Section Manager* (OSM), and a set of *Virtual Processes Managers* (VPMs, each of them running a set of Virtual Processes). The number of VMPs gives the actual parallelism degree of a parmod instance. Also, a number of processes are devoted to application QoS control, e.g. a *Module Adaptation Manager* (MAM), and an *Application Manager* (AM) [6].

The processes that compose an ASSIST application communicate via ASSIST support channels. These can be implemented on top of a number of grid middleware communication mechanisms (e.g. shared memory, TCP/IP, Globus, CORBA-IIOP, SOAP-WS). The suitable communication mechanism between each pair of processes is selected at launch time depending on the mapping of the processes.

2.3 Towards fully grid-aware applications

ASSIST applications can already cope with platform heterogeneity [2], either in space (various architectures) or in time (varying load) [6]. These are definite features of a grid, however they are not the only ones. Grids are usually organised in sites on which processing elements are organised in networks with private addresses allowing only outbound connections. Also, they are often fed through job schedulers. In these cases, setting up a multi-site parallel application onto the grid is a challenge in its own right (irrespectively of its performance). Advance reservation, co-allocation, multi-site launching are currently hot topics of research for a large part of the grid community. Nevertheless, many of these problems should be targeted at the middleware layer level and they are largely

independent of the logical mapping of application processes on a suitable set of resources, given that the mapping is consistent with deployment constraints.

In our work, we assume that the middleware level supplies (or will supply) suitable services for co-allocation, staging and execution. These are actually the minimal requirements in order to imagine the bare existence of any non-trivial, multi-site parallel application. Thus we can analyse how to map an ASSIST application, assuming that we can exploit middleware tools to deploy and launch applications [11].

3 Introduction to performance evaluation and PEPA

In this section, we briefly introduce the Performance Evaluation Process Algebra PEPA [17], with which we can model an ASSIST application. The use of a process algebra allows us to include the aspects of uncertainty relative to both the grid and the application, and to use standard methods to easily and quickly obtain performance results.

The PEPA language provides a small set of combinators. These allow language terms to be constructed defining the behaviour of components, via the activities they undertake and the interactions between them. We can for instance define constants, express the sequential behaviour of a given component, a choice between different behaviours, and the direct interaction between components. Timing information is associated with each activity. Thus, when enabled, an activity $a = (\alpha, r)$ will delay for a period sampled from the negative exponential distribution which has parameter r. If several activities are enabled concurrently, either in competition or independently, we assume that a *race condition* exists between them.

The dynamic behaviour of a PEPA model is represented by the evolution of its components, as governed by the operational semantics of PEPA terms [17]. Thus, as in classical process algebra, the semantics of each term is given via a labelled *multi-transition* system (the multiplicity of arcs are significant). In the transition system a state corresponds to each syntactic term of the language, or *derivative*, and an arc represents the activity which causes one derivative to evolve into another. The complete set of reachable states is termed the *derivative set* and these form the nodes of the *derivation graph*, which is formed by applying the semantic rules exhaustively. The derivation graph is the basis of the underlying Continuous Time Markov Chain (CTMC) which is used to derive performance measures from a PEPA model. The graph is systematically reduced to a form where it can be treated as the state transition diagram of the underlying CTMC. Each derivative is then a state in the CTMC. The *transition rate* between two derivatives P and Q in the derivation graph is the rate at which the system changes from behaving as component P to behaving as Q. Examples of derivation graphs can be found in [17].

It is important to note that in our models the rates are represented as random variables, not constant values. These random variables are exponentially distributed. Repeated samples from the distribution will follow the distribution

and conform to the mean but individual samples may potentially take any positive value. The use of such distribution is quite realistic and it allows us to use standard methods on CTMCs to readily obtain performance results. There are indeed several methods and tools available for analysing PEPA models. Thus, the PEPA Workbench [16] allows us to generate the state space of a PEPA model and the infinitesimal generator matrix of the underlying Markov chain. The state space of the model is represented as a sparse matrix. The PEPA Workbench can then compute the steady-state probability distribution of the system, and performance measures such as throughput and utilisation can be directly computed from this.

4 Performance models of ASSIST applications

PEPA can easily be used to model an ASSIST application since such applications are based on stream communications, and the graph structure deduced from these streams can be modelled with PEPA. Given the probabilistic information about the performance of each of the ASSIST modules and streams, we then aim to find information about the global behaviour of the application, which is expressed by the steady-state of the system. The model thus allows us to predict the run-time behaviour of the application in the long time run, taking into account information obtained from a static analysis of the program. This behaviour is not known in advance, it is a result of the PEPA model.

4.1 PEPA model and performance results

The technical report [1] exposes in details how to model an ASSIST application with PEPA in order to optimise the static mapping of the application. We give in this paper only the general ideas of the approach, and we focus on the on-going and future work, while technical results can be found in the report.

Each ASSIST module is represented as a PEPA component, and the different components are synchronised through the streams of data to model the overall application. The performance results obtained are the probabilities to be in either of the states of the system. From this information, we can determine the bottleneck of the system and decide the best way to map the application onto the available resources.

The PEPA model is generated automatically from the ASSIST source code, during a pre-compilation phase. This task is simplified thanks to some information provided by the user directly in the source code, and particularly the rates associated to the different activities of the PEPA model.

This approach has been introduced on an example of Data Mining classification algorithm [18]. The structure of the application can be represented as a graph, where the ASSIST modules are the nodes and the data streams the arcs. The graph representing this application is displayed in Fig. 2 ❶. The algorithm is designed according to the Divide&Conquer paradigm; all algorithms following the same paradigm could be similarly implemented. It is implemented by means

of four modules: the `start` module is generating the inputs, while the `end` module is collecting outputs. Modules `DC_c45` and `CS_c45` represent the core of the algorithm. The `DC_c45` drives both Divide and Conquer phases: it outputs data items, which are obtained from the split (Divide) or join (Conquer) of input stream items. The `CS_c45` module behaves as "co-processor" of the first module: it receives a data item and sorts it out in such a way that it can be split trivially. Notice that this last module is the only computationally intensive module, while the three others are not.

We have studied the behaviour of the application when mapped in two different ways.

- In the first case (Fig. 2 ❷), we map the loop onto the same cluster, and the `start` and `end` modules onto another. This means that communications on streams s1 and s4 are slow, while they are fast on s2 and s3.
- In the second case (Fig. 2 ❸), we split the loop into two parts, thus `start`, `end` and `DC_c45` are on the same cluster while the computational part `CS_c45` is on the other cluster by itself. In this case, communications on s1 and s4 are fast, while they are slow on s2 and s3.

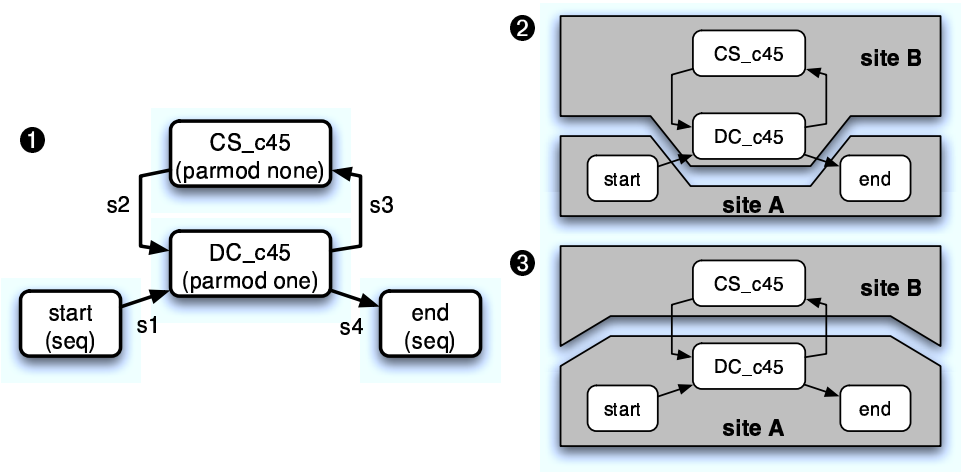


Fig. 2. Graph representation of the classification algorithm (❶) and two different multi-site deployments (❷, ❸).

The results allow us to determine the most efficient mapping, which is the first one, since in this case most of the time is spend in the computational part of the application. The performance results in the second case show that lot of time is spent in the non-computationally intensive modules. This shows that the computational intensive module is constantly waiting for data.

Notice however that it is up to the user to decide which mappings to study, and an exhaustive study of all mappings is probably useless and may cost a lot. The performance models help then to determine, between the mappings chosen by the user, which one is the best.

4.2 Alternative approach to this analysis

The main aim of the analysis performed on the classification algorithm was to compare alternative mappings. In fact, communication and computation rates already include mapping peculiarities (speed of individual links and processors). With the same technique it is also possible to conduct a *predictive analysis*.

Rates are assigned to the PEPA model solely on the basis of the application logical behaviour, assuming uniform speed of connections and processors. In this way, the result of the analysis is not representing a particular mapping, but it rather highlights individual resources (links and processors) requirements, that are used to label the application graph.

These labels represent the expected relative requirements of each module (stream) with respect to other modules (streams) during the application run. In the case of a module the described requirement can be interpreted as the aggregate power of the site on which it will be mapped. On the other hand, a stream requirement can be interpreted as the bandwidth of the network link on which it will be mapped. The relative requirements of parmods and streams may be used to implement mapping heuristics which assign more demanding parmods to more powerful sites, and more demanding streams to links exhibiting higher bandwidths. Whether a fully automatic application mapping is not required, modules and streams requirements can be used to drive a user-assisted mapping process.

Moreover, each parmod exhibits a structured parallelism pattern (a.k.a. skeleton). In many cases, it is thus possible to draw a reliable relationship between the site fabric level information (number and kind of processors, processors and network benchmarks) and the expected aggregate power of the site running a given parmod exhibiting a parallelism pattern [5, 4, 9]. This may enable the development of a mapping heuristic, which needs only information about sites fabric level information, and can automatically derive the performance of a given parmod on a given site.

4.3 Future work

The approach described here considers the ASSIST modules as blocks and does not model the internal behaviour of each module. A more sophisticated approach might be to consider using known models of individual modules and to integrate these with the global ASSIST model, thus providing a more accurate indication of the performance of the application. At this level of detail, distributed shared memory and external services (e.g. DB, storage services, etc) interactions can be taken into account and integrated to enrich the network of processes with dummy nodes representing external services. PEPA models have already been developed for pipeline or deal skeletons [8, 9], and we could integrate such models when the parmod module has been adapted to follow such a pattern.

Analysis precision can be improved by taking into account historical (past runs) or synthetic (benchmark) performance data of individual modules and their communications. This kind of information should be scaled with respect to the

expected performances of fabric resources (platform and network performances), which can be retrieved via the middleware information system (e.g. Globus GIS).

5 Conclusions

In this paper we have presented a way to automatically generate PEPA models from an ASSIST application with the aim of improving the mapping of the application. This is an important problem in grid application optimisation.

It is our belief that having an automated procedure to generate PEPA models and obtain performance information may significantly assist in taking mapping decisions. However, the impact of this mapping on the performance of the application with real code requires further experimental verification. This work is ongoing, and is coupled with further studies on more complex applications.

This ongoing research should be performed between two CoreGRID partners: the ISTI CNR in Pisa, Italy (WP3 - Programming Model), and the ENS (CNRS) in Lyon, France (WP6 - Institute on Resource Management and Scheduling).

Acknowledgments

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

References

1. M. Aldinucci and A. Benoit. Automatic mapping of ASSIST applications using process algebra. Technical report TR-0016, CoreGRID, October 2005.
2. M. Aldinucci, S. Campa, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, and C. Zoccolo. Targeting heterogeneous architectures in ASSIST: Experimental results. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *10th Intl Euro-Par 2004: Parallel and Distributed Computing*, volume 3149 of *LNCS*, pages 638–643, Pisa, Italy, August 2004. Springer Verlag.
3. M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer Verlag, January 2006.
4. M. Aldinucci, M. Danelutto, J. Dünnweber, and S. Gorlatch. Optimization techniques for skeletons on grid. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*, volume 14 of *Advances in Parallel Computing*. Elsevier, October 2005.
5. M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST Grid-aware components. In *Proc. of Euromicro PDP 2006: Parallel Distributed and network-based Processing*. IEEE, 2006. To appear.

6. M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, volume 3648 of *LNCS*, pages 771–781, Lisboa, Portugal, August 2005. Springer Verlag.
7. F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2005.
8. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Evaluating the performance of skeleton-based high level parallel programs. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS, pages 299–306. Springer Verlag, 2004.
9. A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Scheduling skeleton-based grid applications using PEPA and NWS. *The Computer Journal*, 48(3):369–378, 2005.
10. M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.
11. M. Danelutto, M. Vanneschi, C. Zoccolo, N. Tonellotto, R. Baraglia, T. Fagni, D. Laforenza, and A. Paccosi. Hpc application execution on grids. In *Dagstuhl Seminar Future Generation Grid 2004*, CoreGRID series. Springer, 2005. To appear.
12. J. Dünnweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288–294. IEEE Computer Society Press, September 2004.
13. J. Dünnweber, S. Gorlatch, M. Aldinucci, S. Campa, and M. Danelutto. Behavior customization of parallel components application programming. Technical Report TR-0002, Institute on Programming Model, CoreGRID - Network of Excellence, April 2005.
14. I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, Fall 2001.
15. I. Foster and C. Kesselmann, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, December 2003.
16. S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *Proc. of the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, number 794 in LNCS, pages 353–368, Vienna, May 1994. Springer-Verlag.
17. J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.
18. J.R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kauffman, 1993.
19. S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *International Journal Computation and Currency: Practice and Experience*, 2005. To appear.
20. R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency & Computation: Practice & Experience*, 2005.
21. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

Towards an abstract model for grid computing *

A. Stewart[1], J. Gabarró[2], M. Clint[1], T.Harmer[1], P. Kilpatrick[1], R.Perrott[1]

School of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland[1] and
Dep. LSI, Universitat Politècnica de Catalunya, Jordi Girona, 1-3, Barcelona 08034, Spain[2].

Abstract. In this paper an abstract model for component-based grid computing that supports both heterogeneity and dynamicity is proposed. First, a generic definition of a grid site is given. Second, a programming model, ORC (see [3, 4]), is used to describe an applications programmer's interaction with grid sites. ORC offers an abstract means of expressing and reasoning about aspects such as resource allocation, non-responsive hardware and execution monitoring. A number of simple ORC examples, which illustrate how the grid may be navigated and how sets of components may be placed for execution on appropriate sites, are developed.
Keywords. Abstract Grid Model, ORC, Components, Constraints.

1 Introduction

The grid [2] provides an infrastructure for acquiring computing resources. From an application programmer's perspective, utilisation of grid hardware (and software) has two novel features: *heterogeneity* (both hardware and software) and *dynamicity*. Hardware heterogeneity influences the performance of a software component whose efficiency may be heavily architecture dependent. In order to achieve high performance a user may, by means of a *constraint*, specify that a component is to be executed on certain types of grid architectures – for example, a numerical method may only perform well on shared memory architectures with large cache memories. In addition, constraints may be used to specify time and cost bounds for a computation. Software heterogeneity is concerned with the construction of systems from disparate components [1]. For example, it may be the case that a piece of software can only be executed under certain types of operating system, or that it must utilise a particular external database. A user may specify such software dependencies by means of an *external interface*. Constraints and interfaces provide means of selecting appropriate grid sites on which to place a software component for execution. Dynamicity on the grid arises in a number of contexts – for example: the availability of grid sites and network links; resources available per site; wait time for resource acquisition per site; and bandwidth per site. To place a component on a site for execution a user may

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

engage in a dialogue with various grid sites in order to determine what resources are currently available.

In this paper an abstract model for grid computing that supports both heterogeneity and dynamicity is proposed. In §2 definitions of generic web sites and components are given. The definition of site incorporates "yellow pages" information directories, service providers, sites that execute user-defined components and even "user sites" that simply place a component on the grid for execution. Generic sites must contain managers which interact with the rest of the grid. For example, the manager of a user site whose only action is to submit sets of components to the grid for execution must engage in a (series of) dialogue(s) with the grid in order to decide where to place the components. In this paper the (web) orchestration language ORC [3, 4] is used to specify (in an abstract fashion) grid dialogues. In §3 a brief overview of ORC is presented. In §4 simple examples of grid orchestrations are given.

2 Grid Component Model

A grid is a collection of sites. A site is a sextuple comprising a unique name (N), a set of components, or jobs, (C), a collection of services (S) that can be utilised by users, a directory (ID) providing information about grid sites, an engine (E) which has the potential to execute components to produce results, and a local manager (M).

$$gridsite == (N\ ,\ C\ ,\ S\ ,\ ID\ ,\ E\ ,\ M)$$

In a particular site only some of these fields may be instantiated. For example, a "yellow pages" site may contain only a site name, an information directory and a manager, as shown:

$$yellow\_pages = (yp, \_, \_, id, \_, m1)$$

Here id provides information about other sites (for example, the set of sites which offer the capability to execute FORTRAN 90 programs). However, as the grid is a dynamic entity, this information may become obsolete or unreliable after a period of time. Manager $m1$ is responsible for interacting with other grid sites and updating the directory as appropriate. A supercomputer centre offering services, S, (compilers, operating systems, standard libraries such as SCALAPAK etc.) will typically contain a set of jobs awaiting execution, J, a supercomputer, $super1$, for generating results, and a manager. This may be expressed as:

$$super\_computer\_centre = (scc, J, S, \_, super1, m2)$$

The manager $m2$ interacts with users, accepts jobs for execution, manages the "queue" of jobs and returns results to users. Users wishing to submit software artifacts for execution on the grid can themselves be regarded as sites with components and managers.

$$user = (u, C, \_, \_, \_, m3)$$

Here the manager $m3$ is software for placing a set of components C on the grid for execution.

2.1 Constraints

Constraints are statements specifying hardware, performance, cost and network requirements. These have two forms: *minimum* constraints and *value* constraints. A minimum constraint is a set of requirements that *must* be satisfied by any site which is delegated to execute the associated component. A minimum constraint can be modelled abstractly as a predicate. For example, the constraint that a grid machine should have at least 10 processors and a communication link with bandwidth of at least 10^9 bytes/sec may be expressed as:

$$MC \triangleq p \geq 10 \wedge b \geq 10^9$$

For a particular site the parameters p and b may be instantiated, allowing the predicate to be evaluated.

A minimum constraint provides a means of deciding whether or not a site can execute a component. However, it is useful also to be able to determine the "best" site on which to execute a component. A value constraint is an expression with free variables. The expression can be instantiated by individual sites (with the free variables being replaced by values). For example, consider a component, c, with an associated constraint

$$VC \triangleq 10^{10}/ct + b$$

Here ct is a unit computational cost and b is the available bandwidth. VC may be used to measure the "quality" of performance of sites.

2.2 Components

A component is a sextuple comprising an external interface (a set of component names E), an output component name (o), functionality (f), data (d), a minimum constraint (a predicate MC) and a value constraint (an expression VC).

$$component == (E\ ,\ o\ ,\ f\ ,\ d\ ,\ MC\ ,\ VC)$$

The exernal interface of a component c defines its dependencies such as input data sources and utilised service components. The output component, o, is simply an output file. Functionality may be supplied as a combination of program code and service invocations. Alternatively, a component may be used simply to store data.

Example 1. Consider a user who wishes to execute a FORTRAN program, F, on the grid using a file, I, for input to produce an output component, R. The user may construct a data component, *indata*, and a program component, FP:

$$\begin{aligned} indata &== (\_, \_, \_, I, \_, \_) \\ FP &== (\{Fortran90Compiler, indata\}, R, F, \_, MC, VC) \end{aligned}$$

where MC and VC are the constraints (as defined previously). The components FP and *indata* may be sent to, and executed on, any site offering the service *Fortran90Compiler* and satisfying constraint MC. □

2.3 Services

A component that is made available for general use by a grid site is called a service. Activation of a service, sv, by a user results in the future execution of sv. Typically, execution will be performed on the local site. However, a grid site may offer services without having a local engine with which to perform computation. In such a situation a site may acquire an appropriate service from a third party.

The set of services offered by a site may vary with time. User defined components (i.e. software developed by applications programmers) may utilise services through invocation.

Example 2. Consider the following user-defined component:

$$es \quad == (\{eigensolver\}, o, eigensolver(d), d, MC, VC)$$

Here functionality is provided by invocation of the service *eigensolver*. A site that accepts component es for execution implicitly agrees to the use of its eigensolver service. □

A request for a service may not be granted by a site. The procedure for acquiring a service involves both a provisional booking operation, *reserve*, and an actual sevice invocation, $confirm$ (based on the approach taken in ORC [4]). A job j may be placed on a grid site s by a user (or a manager) u through the operation *s.reserve(j,u)*. If s agrees, in principle, to handle the job then it will send a provisional reference number to u. Invocation of the operation $confirm$ (with a valid reference) places j on s awaiting execution.

2.4 Directories

A site directory provides information about services and hardware offered by grid sites. The information provided by directories allows users to navigate the grid and to extract information which allows components to be placed appropriately. Local site information is considered reliable whereas information about external sites may be obsolete. Directories may provide:

1. *local* site information: for example, the set of local services offered by site s is given by *s.services* and the position of component c in the execution queue of s is given by $s.queuepos(c)$. A particularly useful function is *s.can_execute*, which takes a component, c, as argument and, if s can execute the component (that is, it can supply the required software interface and meet the hardware constraints), returns a pair comprising the site name s and the result of evaluating the constraint of c.

2. *grid wide* information: for example site s may be asked to provide the set of all grid sites which can execute a component c ($s.all\_can\_execute(c)$) [1].

2.5 Engines and execution of components

A set of components C, comprising a program and input data, can be executed on a single site s if the site provides an appropriate range of services and also meets all of the component constraints. A set of components that is accepted for execution (a job) is placed on a queue. An engine is a function which transforms such a set of components into an output component:

$$engine : Set(Component) \rightarrow Component$$

Example 3. Placement of the component set $\{FP,\ indata\}$ (see example 1), by a user u, on an appropriate site s creates a job awaiting execution by the site engine. Execution of the job generates an output component R. R may be returned to u or it may be stored while awaiting collection. □

This simple example may be generalised to the case where several engines work in concert to produce results, by adding the notions of input and output streams. In this paper input and output is restricted to files of data.

2.6 Managers

In the definition of grid site above sets of jobs, sets of available services and information directories are *passive*. The site manager interacts with the grid (and with applications programmers) and controls dynamic behaviour. In particular, a manager controls:

- acceptance of new jobs for execution;
- the range of services currently offered;
- the updating of information directories through searches; and
- the job queue (which may involve placing jobs elsewhere).

A user interacts with a site by means of calls to specific functions (such as $s.all\_can\_execute$ (to navigate) or $s.reserve$ and $s.confirm$ to acquire services). In simple situations the manager can decide how to respond using the local site state; however, in general, a manager may also generate calls to third-party sites (with possible side-effects). A user is oblivious to all such secondary communication. This kind of interaction is exactly that provided by the orchestration language ORC [3, 4].

[1] Note that the information supplied by one site s about another may be incorrect.

3 ORC: a language for site orchestration

A brief summary of the language ORC is given here – see [3, 4] for a complete description. ORC is based on the concept of a site call. In ORC all operations must be realised as site calls (e.g. there are no in-built arithmetic operations - addition of x and y may be simulated by the site call $add(x, y)$). In general a site call M may update the recipient site which, in turn, may call other sites and reply. A fundamental concept of ORC is that a site call may fail (i.e. the sender may not receive a reply). This may be due to a faulty network (either the outgoing or incoming message may fail) or even to the recipient site being down. There are some special sites:

- 0 never responds (0 can be used to terminate expressions);
- if b returns a signal if b is true and remains silent otherwise;
- $RTimer(t)$, always responds after t time units;
- let always returns (publishes) its argument.

In addition, calls made to the generic grid site defined in § 2 are considered.

ORC site calls may be orchestrated by means of expressions. The simplest expression is a site call, possibly with parameters. More complex expressions can be defined as follows, where $E1$ and $E2$ are expressions:

1. operator $>$ (sequential composition)
   $E1 > x > E2(x)$ evaluates $E1$, receives a result x, calls $E2$ with parameter x. If $E1$ produces two results, say x and y, then E2 is evaluated twice, once with argument x and once with argument y. The abreviation $E1 >> E2$ is used for $E1 > x > E2$ when evaluation of $E2$ is independent of x.
2. operator $\mid$ (parallel composition)
   $(E1 \mid E2)$ evaluates $E1$ and $E2$ in parallel. Both evaluations may produce replies. Evaluation of the expression returns the merged output streams of $E1$ and $E2$.
3. where (asymmetric parallel composition)
   $E1$ where $x :\in E2$ begins evaluation of both $E1$ and $x :\in E2$ in parallel. Expression $E1$ may name x in some of its site calls. Evaluation of $E1$ may proceed until a dependency on x is encountered; evaluation is then delayed. The first value delivered by $E2$ is returned in x; evaluation of $E1$ can now proceed and the thread $E2$ is halted.

3.1 Site Failure

In the case of site failure (observed as silence) a user may remain waiting for a response (non-termination). Silence may be temporary, due to a delay in response from the target site, or permanent. Evaluation of some ORC expressions *must* succeed. For example, the expression

$$Terminate \triangleq \text{let(s)} \ \text{where} \ s :\in \{N \mid Rtimer(100) >> let(stop)\}$$

has one thread $Rtimer(100) >> let(stop)$ comprising local site calls only. These calls must succeed and, thus, evaluation of the expression must terminate. However, this is not the case for an arbitrary site call. Grid programs should react when a response, which thay are awaiting, is not forthcoming after some interval. Typically, if there is no response to a site call after a specified period then an exception handling routine will be invoked. One way of dealing with site calls that do not respond is to repoll the site. Thus, instead of conducting a single poll at one instant, polls may be carried out at regular or irregularly spaced intervals. Two such polls, $Poll^*$ and $TPoll$, which repeatedly instantiate an ORC expression, E, are defined below:

$$Poll^*(E) \triangleq \mathsf{let(r)} \ \ \mathsf{where}\ r :\in \{\,|_{t\in\mathcal{N}}\ Rtimer(t) \gg E\}$$

Here E is instantiated (infinitely often) at regularly spaced intervals. Each thread in $Poll^*$ waits indefinitely for a reply.

In many situations a user may wish to poll repeatedly until a response is received. With the time poll, $TPoll$, multiple threads are generated, but threads which do not respond after t time units are ignored. In effect, old threads are "cancelled" before new threads are invoked. The arguments f and t may be used in the definition below to vary the interval between polls:

$TPoll(E,t,f) \triangleq$
  $\mathsf{if}\ flag \gg let(s) \,|\, \mathsf{if}\ \neg flag \gg\ TPoll(E, f(t), f)$
  $\mathsf{where}\ (flag, s) :\in \{E > s > let(true, s) \,|\, Rtimer(t) \gg let(false, failure)\}$

Here the site if returns its argument if it is true and remains silent otherwise. Only one of the component expressions in the parallel statement

$$(\ \mathsf{if}\ flag \ldots) \,|\, (\ \mathsf{if}\ \neg flag \ldots)$$

can become active – in this case either $TPoll(E, f(t), f)$ or $let(s)$. Evaluation of $TPoll(E, 1, \lambda x.x)$ generates a sequence of polls separated by 1 time unit while evaluation of $TPoll(E, t, \lambda x.x + x)$ generates a sequence of polls at times 0, t, 2t, 4t, etc. A new thread is activated at time $f(t)$ if the earlier thread fails to respond in the interval $t, \ldots, f(t)$.

4 Component placement: examples

4.1 Site Selection

Suppose that a user (or manager) knows a set of names of grid sites, $\mathcal{F} = \{s_1, \ldots, s_n\}$. Consider the selection of an appropriate site on which to place a component, c, for execution.

$$Select1(c, \mathcal{F}) \triangleq \{let(s)\ \mathsf{where}\ (s, v) :\in (\,|_{s_i\in\mathcal{F}}\quad s_i.can\_execute(c))\}$$

Select1 generates a call to each of the sites in $\mathcal{F}$ to determine which are suitable for the placement of c; the first site to respond returns its site name, s, and the

value, v, of V on s – evaluation of *Select1* publishes s. Note that if none of the sites can execute c then the evaluation of *Select1* will not terminate.

The first site to respond may not be the best location for c. The best site on which to place c is the one with the largest local value (*measure*) of V. This best site may be found by passing pairs of sites and corresponding measures to a local store z using the local site call $z.pass$; the pair which has the highest measure may be retrieved by the call $z.highest$. The expression *Best*, below, constructs multiple threads which send a stream of site information to z; after calling $z.pass$ each thread is terminated.

$$Best(c, \mathcal{F}) \triangleq z.null >> (\,|_{s_i \in \mathcal{F}}\ s_i.can\_execute(c) > (s, v) > z.pass(s, v) >> 0)$$

Here z is initialised by means of the site call $z.null$. The best site available after t time units may be selected as follows:

$$Select2(c, \mathcal{F}, t) \triangleq let(s) \text{ where } (s, v) :\in (Best(c, \mathcal{F}) \,|\, Rtimer(t) >> z.highest)$$

Evaluation of *Select2* is guaranteed to terminate; however, when none of the active sites in $\mathcal{F}$ can execute c, a null site name is published. A user who polls the set of sites $\mathcal{F}$ and does not receive in reply a valid site name may wish either to poll a larger set of sites or alter the given constraint set. A larger set of potential sites may be found using grid information directories.

$$Trawl(c, \mathcal{F}) \triangleq y.empty >> (\,|_{s_i \in \mathcal{F}}\ \ s_i.all\_can\_execute(c) > \mathcal{G} > y.union(\mathcal{G}) >> 0)$$

Here each thread in $Trawl$ determines a set of sites with the potential to execute c. The sets are combined by distributed union and the result is held in y. Distributed union is realised through a stream of *local* site calls $y.union$ each of which has the side effect of combining its set argument with y. The site call $y.empty$ is used to initialise y. After t time units the set currently stored by y is extracted using the operation $y.get$:

$$Select3(c, \mathcal{F}, t) \triangleq Select2(c, \mathcal{G}, t) \text{ where } \mathcal{G} :\in (Trawl(c, \mathcal{F}) \,|\, Rtimer(t) >> y.get)$$

Note that the set of sites $\mathcal{G}$ found by browsing information directories may not be valid. Direct contact is made with each of these sites using the expression $Select2(c, \mathcal{G})$ to verify that it can be used to place c.

A user may, in the event of not being able to find a suitable site to place c, weaken the associated component constraint. Let c' be a component which is the same as c except that it has a weakened constraint. A selection mechanism which first tries to find where to place c and, if unsuccessful, then tries to find where to place c' is:

$Select4(c, \mathcal{F}, t) \triangleq$

(if $\neg$null(s) $let(s)$ | if null(s) $Select3(c', \mathcal{F}, t)$) where $(s, v) :\in Select3(c, \mathcal{F}, t)$

This strategy can be repeated to allow the constraint to be further weakened.

It may happen that a user tries to select a site to place a component when the grid network is congested – in such circumstances responses to site calls may not be delivered. To make site selection more robust, it may be desirable to use a form of repeated polling (as in §3.2).

4.2 Placement

Assume that an ORC expression $Select(C, \mathcal{F})$ returns a site, in F, that can be used to execute the set of components C if such a site exists and remains silent otherwise. A variety of types of placement are considered below.

4.2.1 Single site placement:
A single site job placement by user u may be made as follows:
$Place(u, \{c\}, \mathcal{F}) \triangleq let(r)$ where
$r :\in (Select(\{c\}, \mathcal{F}) > s > s.reserve(\{c\}, u) > ref > s.confirm(\{c\}, ref)$
This program is not robust and may fail if s does not agree to the proposed placement or if the network is currently faulty. The placement function can be made more robust using techniques similar to those used in the Site Selection section.

4.2.2 Multiple site, independent components placement:
Two independent components c_1 and c_2 can be placed and executed independently as follows:
$Independent(u, \{c_1, c_2\}, \mathcal{F}) \triangleq let(r1, r2)$
where $r1 :\in Place(u, \{c_1\}, \mathcal{F})$ where $r2 :\in Place(u, \{c_2\}, \mathcal{F})$

4.2.3 Multiple site, dependent components placement:
Consider two components c_1 and c_2 where c_1 has output component name f (i.e. an output file) and c_2 has an external (input) interface $\{f\}$. One method of placing c_1 and c_2 is first to place c_1, await the return of the component f, and then place c_2 and f:

$Place2(u, \{c_1, c_2\}, \mathcal{F}) \triangleq let(r)$ where
$r :\in (Place(u, \{c_1\}, \mathcal{F}) > f > Place(u, \{c_2, f\}, \mathcal{F}))$
Here the user's site acts as a hub for controlling the orchestration (i.e. the data file is returned to the user for subsequent placement). A significant delay could occur between receiving f and finding a suitable site on which to place c_2. An alternative placement approach, which avoids passing f to u, is:

$Place3(u, \{c_1, c_2\}, \mathcal{F}) \triangleq Place(u, \{c_2, f\}, \mathcal{F})$ where $f :\in Place(u, c_1, \mathcal{F})$

Evaluation of this expression tries to place the two jobs simultaneously. However, the expression $Place(u, \{c_2, f\}, \mathcal{F})$ cannot be evaluated until f is available. This restriction may be avoided as follows: f is a data component and so has no effect on the selection or reservation of a site for c_2. A site for placing c_2 *alone* can be

reserved prior to the acquisition of f:

$Place4(u, \{c_1, c_2\}, \mathcal{F}) \triangleq$

$Select(\{c_2\}, \mathcal{F}) > s > s.reserve(\{c_2\}, u) > ref > s.confirm(\{c_2, f\}, ref)$

where $f :\in Place(u, \{c_1\}, \mathcal{F})$

Note that f is introduced only at the confirmation stage.

4.3 Monitoring remote execution

Consider again the single site placement where a component c has been placed on a site s by a user. A delay may ensue before computing resources become available for executing c. The user may monitor the position of c on a queue as follows:

$Monitor(c, s) \triangleq s.queue(c) > n >$

(if $n = 0$ $let(0)$ | if $n > 0$ $let(n)$ $>>$ $RTimer(t)$ $>>$ $Monitor(c, s)$)

Evaluation of this expression will produce a stream of natural numbers showing the position of c in s's queue; evaluation of the expression terminates when c reaches the head of the queue.

5 Discussion

In this paper an extended example has been developed which shows how an applications programmer might interact with the grid. The structure proposed for modelling grid sites is used, primarily, as a basis for constructing ORC expressions. The purpose of the paper is to propose to the grid community that ORC is an appropriate vechicle for describing the orchestration of grid resources.

It should be noted that there are two kinds of grid dialogue. In this paper we have focused on the kinds of activity that an applications programmer might wish to engage in. On the other side of the coin is the internal behaviour of sites which respond to user requests (i.e. manager behaviour). This paper does not address internal site issues. Instead an abstract programming model in which grid sites are viewed as black boxes is proposed; from this standpoint the details of how site calls are managed internally is immaterial. However, managers will be obliged to engage, inter alia, in the same kinds of behaviour as have been attributed to users in this paper.

References

1. Caromel, D, Henrio, L.: *A Theory of Distributed Objects.* Springer, 2005.
2. Foster, I., Kesselman, C.: *The GRID: Blueprint for a new computer infrastructure.* Morgan Kaufmann, 1999.
3. Hoare, T., Menzel, G., Misra J.: A Tree Semantics of an Orchestration Language. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series Marktoberdorf, Germany, 2004.
4. Misra J.: Computation Orchestration: A basis for a wide-area computing. In M. Broy, editor, *Proc. of the NATO Advanced Study Institute, Engineering Theories of Software Intensive Systems*, NATO ASI Series Marktoberdorf, Germany, 2004.

Improving transparency of a distributed programming system

Boris Mejías[1], Raphaël Collet[1], Konstantin Popov[2], and Peter Van Roy[1]

[1] Université catholique de Louvain, Louvain-la-Neuve, Belgium
{bmc, raph, pvr}@info.ucl.ac.be
[2] Swedish Institute of Computer Science, Stockholm, Sweden
kost@sics.se

Abstract. Since Grid computing aims at creating a single system image from a distributed system, transparent support for distributed programming is very important. Some programming systems have attempted to provide transparent support for distribution hiding from the programmer several concerns that deal with the network behaviour. Sometimes, this leads to restricted models that attach programmers to the decisions of language designers. This work propose language abstractions to annotate entities specifying their distributed behaviour while keeping the transparency of the distribution support, and a failure model that separates the application's logic from failure handling.

1 Introduction

It is well known that distributed programming is hard, and large research effort has been devoted to simplify it. One of the approach is transparent distribution, where the distributed system appears to the user as a single system image. In such environment, semantics of operations performed in a centralised system remain the same in a distributed one. Even though, distributed programming is plagued with network failures, lack of global time and other problems shown in [1] that make full transparency infeasible. However, transparent distribution can be achieved in several degrees.

In transparent distribution, the consistency of the system is maintained using predefined access architectures and protocols, which are designed depending on the semantics of each type of entity. These distribution strategies cannot be modified by the programmer, who is limited to use the default design. We propose a way to annotate language data structures, here referred as entities. These annotations will allow programmers to choose between several strategies for every entity according to the non-functional requirements of their programs.

Annotating entities in peer-to-peer networks becomes very helpful to keep the consistency of their states. Knowing that the peer that is currently hosting an entity can leave the network at any time, one may annotate it with a migratory strategy. As soon as another peer request access to it, the entity will migrate to the requesting peer, and the original host can safely leave. Regarding distributed garbage collection, a peer can chose a persistent strategy to keep an entity alive even when temporary there is no distributed reference to it.

The platform we are using to test our concepts is based in the Distribution SubSystem (DSS) [4], which is a language independent middleware for efficient distribution support of programming systems. Working in collaboration between SICS and UCL, we have integrated the DSS into the Mozart system[2, 3], which provides a state of the art in transparent distribution. Inspired by the work presented in [5], we also propose a new fault model to improve fault handling in a more factorised way. This fault model, together with the election of right distribution strategies, can strongly help in design and implementation of decentralised application, such as Peer-to-Peer networks and Grid-based applications, where network failures are very common.

Our proposal will be directly applied in the support of the Peer-to-Peer networking library, P2PS[6], but the results is not limited to that. Existing programming models for Grid at best support failure handling at the level of services or components, but not at the granularity level of individual operations and individual data items. Such fine-grained failure handling is important in particular for building high-performance Grid-based applications that must react to changes in the network promptly, without reconfiguring distributed Grid services involved in the application. We believe that this work helps in the achievement of that required granularity.

2 Entity Annotations

Let us consider the example of a cell with a state that can be updated. To maintain its state consistent over the distributed system, several protocols are provided such as "migratory" or "stationary". In the migratory protocol, the state migrates to the site that is performing the operation. In the stationary protocol, the operation moves to the site where the state resides, and only the result of the operation travels back. We may also take into account the algorithm that maintains the distributed references to the entity, to do a proper garbage collection. A programmer could ask for a reference counting or a time-lease based mechanism, or even force the entity to never become garbage. It is also possible to parametrise the communication architecture of the group of sites referring to a given entity.

Currently, there is no way to choose arbitrarily which protocol to use. The decision is made by language designers and programmers are limited to it. We provide a language abstraction to *annotate* the entity deciding its distribution strategy. The following formal semantics were presented in the Nordic Workshop of Programming Theory (NWPT'05) [7], and they represent a strong base for our development of applications. The operational semantics of the abstraction are expressed using reduction rules as

$$\frac{S \,\|\, S'}{\sigma \,\|\, \sigma'}\, C$$

where C is a boolean condition, *S* and σ are the statement and store before the reduction, and *S'* and σ' are the statement and store after the reduction.

Rule (1) defines the semantics to annotate an entity E with the annotation A. The annotation is a keyword with the name of the distribution strategy, such as "migratory", "stationary", "read_write_invalitation", etc. It can also be express as a record of the

form *a(prot:P arch:A gc:G)*, where P represents the protocol, A the communication architecture, and G the garbage collection algorithm.

$$\frac{\{\texttt{Annotate}\ E\ A\}}{\sigma} \Big\| \frac{\texttt{skip}}{annot(E,A) \wedge \sigma} \text{ if } \forall B : \sigma \models annot(E,B) \Rightarrow compat(A,B) \tag{1}$$

The rule allows incremental annotations over an entity as long as they define a relation of *compatibility*. Two annotations are compatible if they do not implies contradiction in the behaviour of each part of the distribution strategy. Then, an entity could be annotated it to have a migratory state and a reference counting algorithm for garbage collection, but, having a migratory state and a stationary state is of course not allowed. The system also provides default annotations to be used when the programmer does not specify any.

3 Operations over an annotated entity

Let us consider now the Exchange operator to read and write the state of a cell in only one step. In a local computation, the operation follows the semantic of rule 2, where E represents the entity, Y is the variable to be unified with the old value of the entity, and Z corresponds to the new value. The entity E is bound to the mutable pointer e, having its state represented with the statement $e{:}W$, where W corresponds to the current value. The operation is reduce to the the binding of Y with the old value W, where Z becomes the new current value of the cell.

$$\frac{\{Exchange\ E\ Y\ Z\}}{E{=}e \wedge e{:}W \wedge \sigma} \Big\| \frac{Y{=}W}{E{=}e \wedge e{:}Z \wedge \sigma} \tag{2}$$

Now we present the consecuence of annotating an entity. We are considering one single store for all the nodes involved in the network. We introduce indexes to specify the location of a particular statement, then, $(X{=}Y)_i$ describes a unification between entities X and Y that takes place in node i. The following rules define the distributed behaviour that implements the chosen protocol. Note that if indexes are removed, semantics remain the same, keeping distribution support transparent.

$$\frac{\{Exchange\ E\ Y\ Z\}_i}{annot(E,A) \wedge E{=}e \wedge (e{:}W)_j \wedge \sigma} \Big\| \frac{(Y{=}W)_i}{annot(E,A) \wedge E{=}e \wedge (e{:}Z)_i \wedge \sigma} \tag{3}$$

$$\text{if } A{=}migratory$$

$$\frac{\{Exchange\ E\ Y\ Z\}_i}{annot(E,A) \wedge E{=}e \wedge (e{:}W)_j \wedge \sigma} \Big\| \frac{(Y{=}W)_i}{annot(E,A) \wedge E{=}e \wedge (e{:}Z)_j \wedge \sigma} \tag{4}$$

$$\text{if } A{=}stationary$$

In rule 3, the entity has been previously annotated as “migratory”. Originally, the state $e{:}W$ resides in node j, and Exchange is invoked at node i. The operation will reduce to the unification of Y with the old value W at the same node where the operation was invoked. Due to the migratory annotation, the new state $e{:}Z$ will also move to node

i. Rule 4 is the equivalent operation having a "stationary" annotation. In this case, the result of the operation will travels back to node i, but the new state will remain in the original note j, because of its stationary strategy.

Note that in both rules, the reduction is a new operation at the invoking site, because the result travels back to the invoker, no matter where the operation of writing the new state is performed. The reduction of a unification will be entailed by the store as a global information, which is consistent with the statement $E{=}e$ of the previous rules.

$$\frac{(Y{=}W)_i \;\Big\|\; skip}{\sigma \;\Big\|\; Y{=}W \wedge \sigma} \text{ if } \sigma \models Y{=}W \tag{5}$$

4 Fault Model

We also propose improvements on the fault model of Mozart presented in [8]. Currently, if the distribution support of a certain entity presents a connection failure, an exception will be raised when an operation is performed and therefore, transparency will be unexpectedly broken. Let us consider the case of a piece of code that was written for a local execution that is used in a distributed program. This code reusing is perfectly reasonable because the system aims at transparency. The problem appears when a network failure is detected. The piece of code triggers a totally unexpected exception, making fault handling very hard to program.

We propose that operations silently block in case of failure. The operation is able to resume if the system recovers from the failure, or it might block forever. Fault handling is done in a concurrent thread which monitors the fault state of the entity, taking the correspondent actions. The advantage of that design is that pieces of local code will not modify their semantics when they are executed in a distributed environment. This improves the factorisation of fault handling. There is no full transparency at this level, but a higher degree is achieved.

Each distributed entity can be in one of three fault states: *ok* (no failure), *tempFail* (temporary failure), and *permFail* (permanent failure). The latter state means that the entity will never recover. Valid transitions between those states are defined by the automaton in Figure 1.

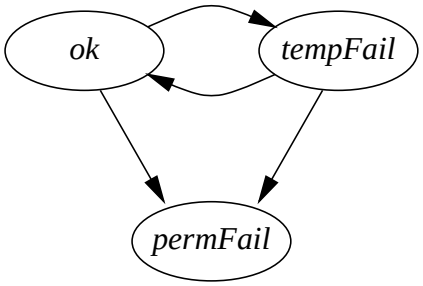


Fig. 1. Fault state transitions

In order to monitor fault states, each entity has a *fault stream*, which is a list of fault states. The system identifies failures in the communication between sites, and reports changes in an entity's fault state by extending its fault stream. Rule (6) models this, with $fs(E, s|sr)$ associating entity E with its fault stream $s|sr$. Rule (7) shows how to get access to the fault stream of an entity. The stream is always prefixed with the current fault state of the entity. Dataflow synchronisation automatically awaken a monitoring thread when the fault state of the monitored entity changes.

$$\frac{S \;\Big\|\; S}{fs(E, s|sr) \wedge \sigma \;\Big\|\; fs(E, s'|sr') \wedge sr{=}s'|sr' \wedge \sigma} \text{ if } s \rightarrow s' \text{ is valid transition} \tag{6}$$

$$\frac{\{\texttt{GetFaultStream}\ E\ S\} \Big\| \quad S{=}s|sr}{fs(E, s|sr) \wedge \sigma \quad \Big\| fs(E, s|sr) \wedge \sigma} \tag{7}$$

This model does not pretend to hide failures. Instead, it offers the possibility to treat them in a concurrent thread. The following code is an example of a thread monitoring a distributed entity. We first get the fault stream of the entity. Then, a thread is launched monitoring the fault stream, meanwhile another thread is performing operations over the entity.

```
FS = {GetFaultStream Entity}
thread {Monitor FS} end
thread <several operations over Entity> end
```

An example of a procedure that monitors the fault stream of any entity follows. The procedure receives the stream and try to do pattern matching with all possible states. If no failure is reported to the stream, the pattern matching will block. Once a fault state is matched, the procedure will take the correspondent action according to each case, and it will continue monitoring the rest of the stream. In the case of the example, temporal failure are treated with a time out.

```
declare
proc {Monitor Stream}
   case Stream of S|Sr then
      case S
      of ok then skip
      [] tempFail then
         {WaitOr Sr.1 TimeOut}
         <doSomething>
      [] permFail then
         <doSomething>
      end
   end
   {Monitor Sr}
end
```

5 Conclusions and Future Work

We have presented language abstractions that improve transparent distribution support. Entity annotation allows programmers to decide the distribution strategy that fits better to each entity, providing a granularity that cannot be achieved at the level of distributed systems based on components. These annotations do not break the transparency of the distribution support, helping to conceive the distributed system as a single image.

Being aware of network failures, which is the main limitation of distribution transparency, we proposed a new failure model to improve modularity and transparency. Failures are reported to a fault stream created per entity. This allows to monitor failures concurrently, avoiding unexpected exceptions in code extended to a distributed execution. This also provides granularity that is not presented in component based systems.

As future work, we will design guidelines to use the more convenient annotations according to each scenario, having first in mind peer-to-peer applications. We also need

to define proper actions in failure handling to keep consistency of the system. Implementation also need to be finished.

6 Acknowledgements

This research is partly supported by the projects CoreGRID (contract number: 004265) and EVERGROW (contract number:001935), funded by the European Commission in the 6^{th} Framework programme, and project MILOS, funded by the Walloon Region of Belgium, Convention 114856.

References

1. Waldo, J., Wyant, G., Wollrath, A., Kendall, S.: A note on distributed computing. In: Mobile Object Systems: Towards the Programmable Internet. Springer-Verlag: Heidelberg, Germany (1997) 49–64
2. Mozart-Oz: The mozart-oz programming system. (*http://www.mozart-oz.org*)
3. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press (2004)
4. Klintskog, E.: Generic Distribution Support for Programming Systems. PhD thesis, KTH Information and Communication Technology, Sweden (2005)
5. Grolaux, D., Glynn, K., Van Roy, P.: A fault tolerant abstraction for transparent distributed programming. [9] 149–160
6. Mesaros, V., Carton, B., Van Roy, P.: P2ps: Peer-to-peer development platform for mozart. [9] 125–136
7. Mejías, B., Collet, R., Van Roy, P.: It's such a fine line between transparent and non-transparent distribution. In: The 17th Nordic Workshop on Programming Theory. (2005) 94–96
8. Van Roy, P.: On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in mozart (1999)
9. Van Roy, P., ed.: Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers. In Van Roy, P., ed.: MOZ. Volume 3389 of Lecture Notes in Computer Science., Springer (2005)

A Vision of Metadata-driven Restructuring of Grid Components

Armin Größlinger and Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau
{groessli,lengauer}@fmi.uni-passau.de

Abstract. Work Package 3 (WP3) of the CoreGRID network of excellence focuses on the design of a standard component model for the European Grid community. Through this network, we have had contact with other research groups in different countries. In this extended abstract we sketch our vision for a Grid component model which has been heavily influenced by the discussion with WP3 partners.

1 Introduction

A variety of component models have been developed, most of them for a specific environment. The Fractal Component Model [BCS04] has an open set of control capabilities, i.e., the capabilities and mechanisms to control, configure and reconfigure a set of components are not fixed in the model, but can be chosen by the component developer. This flexibility ensures that Fractal can be used to implement any feature or extension in an upcoming Grid component model (GCM).

The dynamic nature of Grid environments requires that an application is not a static piece of software, but that the arrangement of the components and the components themselves are adaptable to the environment. Szyperski [Szy02] demands that components are units of depolyment and composition and that they have no externally observable state. This implies that they are opaque, i.e., their "contents" cannot be accessed through their interfaces. In our view of a GCM, components are opaque for their users, i.e., for the application programmer, but the runtime support for the GCM will need to access the contents of components and modify it, or even replace a set of components by a new one which is constructed from the constituents of the components to be replaced.

We hold the opinion that dynamic code generation (which may reuse some of the existing code in the components) and code motion (i.e., the migration of objects/components or parts thereof) is the key to executing Grid applications efficiently. We discuss briefly three prototype scenarios and one real-world scenario to illustrate our point in Section 2 and sketch our vision for a GCM in Section 3.

2 Scenarios

Before we present our idea for a GCM, we sketch some scenarios which are to be understood as minimal examples of the problems our suggested GCM is supposed to solve. To simplify the illustration, these scenarios are given in terms of existing technology (like NFS file servers, X servers, etc.), but the principle applies equally to proper Grid applications.

2.1 Accessing File Servers

In a first scenario, we consider the frequent situation that an application is accessing some remote data source (e.g., an NFS file server) to manipulate data. In a traditional approach, the data is first transferred to the client, then modified by the client and finally sent back to the server. As a simple example consider a file copy operation. Here, the client does not modify the data at all (i.e., it applies the identity function to it), but it is nevertheless sent twice over the network. Figure 1 (a) shows the relevant parts of this scenario. Client and server both interact with network components (e.g., components talking TCP or UDP) and the server uses file system interface components to access its disks. Figure 1 (b) shows a more efficient version of the application with the same semantics. Now the copying takes place solely at the server, so no network traffic arises. Note that the function f the client applies to the data in Figure 1 (a) has been moved to the server in Figure 1 (b).

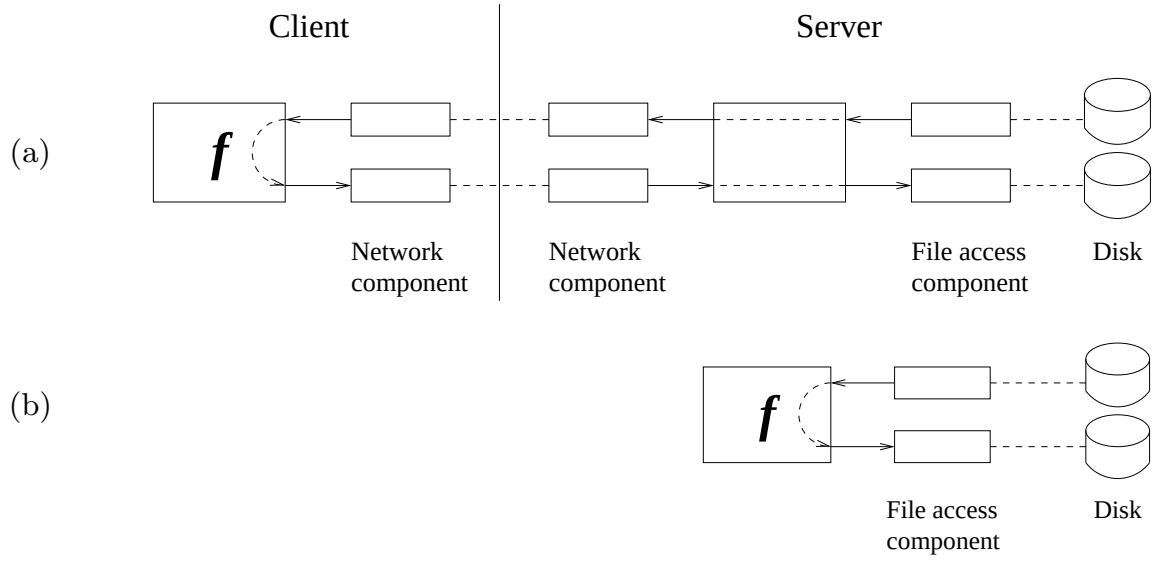


Fig. 1. A file transfer in a client-server scenario

Since detecting the point in the execution of the application when to move code from the client to the server (and finding the right kind of transformation at all) is difficult, another optimisation may seem easier and at least as profitable: using futures. Instead of sending the data to the client and sending it back to

the server, only placeholders are transferred which record the operations to be performed on the data. When the operation completes (more generally, when the effects of the operation must be made visible to the rest of the application or the outside world), an efficient execution plan is computed [YK03]. This can produce an execution equivalent to Figure 1 (b). The problem with this approach is that the placeholders have to be transferred and that not every f at the client can be recorded in the placeholders appropriately. For example, if f is a loop copying the input data to the output stream, like

```
while (!in.eof()) {
    byte b = in.read();
    out.write(b);
}
```

the predicate `in.eof()` (which checks for end of input) cannot be evaluated, unless each call of `in.read()` advances the file pointer of the input stream *on the server*, such that end of input can be detected on the server. Therefore, the call of `in.eof()` nullifies the improvement gained by using futures.

2.2 Distributed Applications and GUIs

As a second scenario, let us consider an application with a graphical user interface. As is common with X Window System (X11) today, an application can be run on one machine (in X11 terms called "client") and display its GUI on a different machine (called "X server" in X11 terms). Running applications over the local Ethernet is rather comfortable most of the time, but with dial-up lines or even DSL (which often has a rather limited upstream bandwidth), applications become sluggish or even unusable. A reproducible effect of an application becoming unusable can be illustrated with the popular Firefox browser (Firefox 1.0.6 on Linux compiled for GTK+ 2.0). Starting Firefox on a remote machine and displaying its windows on an X server, which is connected to the Internet by a standard DSL line with 1MBit upstream and 128kBit downstream, the overall GUI speed is slow, but usable. But if one starts a download, Firefox's download manager (the window displaying the progress of the download) takes several minutes(!) of full capacity upstream transfer to compose itself on the screen. This delay in the GUI also delays the download and Firefox becomes all but usable.

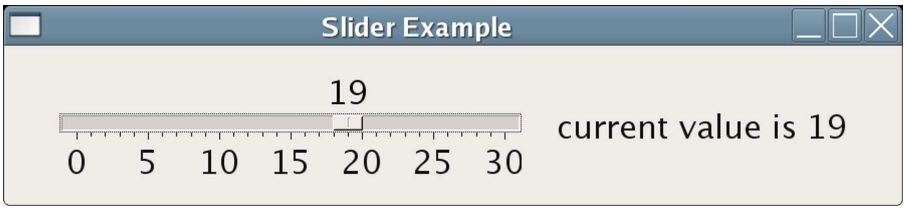


Fig. 2. A GUI with dependent elements: the label depends on the slider

The scenario we discuss here is much simpler but illustrates the main point. Figure 2 shows a tiny part of a GUI. The user can manipulate a slider to select a certain value, and the value is displayed in a label widget. When the user moves the slider, several events (mouse button click, mouse move, mouse button release, etc.) have to be communicated to the X client, where they trigger event handlers. The event handlers cause the slider to be redrawn (requiring communication with the X server) and set the label's text to a new string (namely the new value of the slider). This causes some more communication between the client and the server. To yield a responsive distributed application, the GUI widgets (the slider and the label) have to reside on the server (e.g., moving the slider must not cause drawing requests being sent from the client to the server). In addition, the event handlers for the slider must be moved to the server in order to achieve prompt updates of the label. This is the tricky part in making distributed GUI applications responsive: the right amount of code to be moved has to be identified (we do not want to move code accessing a local file on the client, for example).

2.3 Accessing Data Sources

Our third scenario is a more data-centric scenario. The Grid is not only a computational resource, but also a data source. Suppose an application accesses several databases found on the Grid. The application combines the data and applies filters to it. Some of the filtering may be done before combining the data, so the filter function can be moved from the consumer application to the data source, filter the data there and transfer only the remaining data (this process is similar to the classic relational query optimisation for distributed databases). If it is likely that some filter (or a similar filter) is applied again in the future, caching the filtered data at the data source and/or the consumer application can be profitable. Moving the filter function involves moving code to the data source. The challenge here is to know when to cache data and what data to reuse to answer a query. This implies that components may have to be split into computation and data access parts such that data accesses and filter functions can be treated as separate objects.

2.4 A Real World Scenario

A more realistic scenario, which combines all three prototype scenarios, is as follows. A user starts a Web browser on machine C (the client) and displays it on an X server running on machine X. The browser (or a suitable plugin) is used to play a streaming video from a server V (see Figure 3). With today's X11 technology, the video is transferred from machine V to machine C (because the browser runs on machine C). Then, to display the video frames on the screen, it is sent again over the network to the display at machine X (together with the browser's GUI). This is suboptimal as the video gets transferred twice, once as the video stream and once as the rendering performed by the browser. Since bandwidths between the nodes V, C, and X can be different, it may be necessary to lower the quality of the video at C to cope with a lower bandwidth between

C and X and, even worse, the bandwidth between V and X may be higher than the bandwidth on the path V-C-X, which means that the video is displayed with suboptimal quality. The situation can be improved by exploiting the adaptation described in the three basic scenarios. First, the video is transferred from V directly to X. Second, the GUI of the browser and the video decoding/display code is moved from C to X. And finally, if the bandwidth between V and X is too low to display the video in full quality, the quality (and hence bandwidth) reduction should be performed at V. In addition, if the quality-reduced video is cached at V, it can be reused for future requests. Furthermore, the reduced video can be cached at different sites anticipating future uses of it.

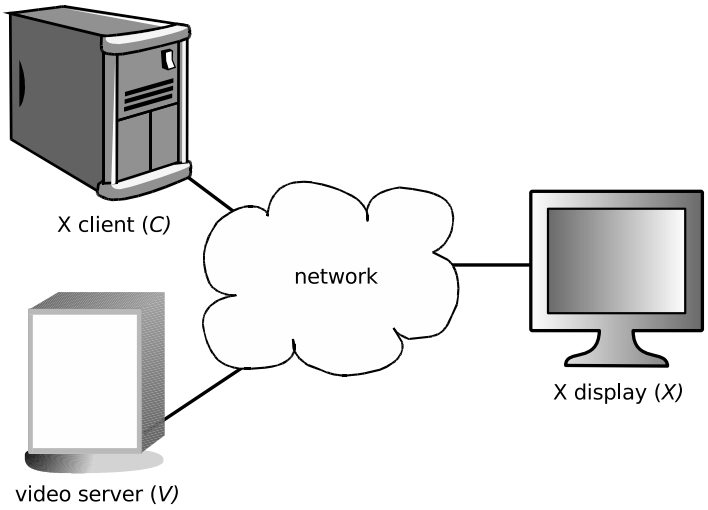


Fig. 3. Remote access to an application and a video stream

3 A Grid Component Model

The scenarios given in Section 2 may not seem very Grid-related. But, in our view, they express some fundamental problems encountered in future Grid applications based on existing technology. If we abstract from the concepts of the NFS server or the Web browser, we can view the scenarios as describing situations in which an application provider offers an application (or application components) with a GUI (symbolised by the Web browser) to a user who uses the application to access a remote data source (symbolised by the video stream).

3.1 Our Discussions with other WP3 Members

Our vision of a GCM comes from several sources. From the Pisa group (led by Marco Danelutto), we have learned about the Grid.it model [Gri04] and its concept of a *manager* which is part of a component and which communicates with

the outside through the non-functional interface of the component. The context of a component can request performance contracts, assign resources to the component, or query the component's status through the manager's interfaces. Our vision presented here leaves the role of the manager mostly open, since we do not want to fix the distribution of the jobs discussed here between the runtime system and the manager at this time.

With Paul Kelly and Olav Beckmann (Imperial College, London) we had several discussions about their work on runtime optimisation using delayed evaluation and self-optimising components [LBK02]. They introduced us to the idea of having metadata associated with components to drive the optimisations.

3.2 Requirements on a Grid Component Model

In our vision of a GCM, two principal requirements should be met by the components and the runtime system. We state these requirements in this subsection and discuss the metadata necessary to enable our principal requirements to be implemented in Subsection 3.3. As we have outlined in the description of our scenarios (Section 2), our optimising transformations base on two fundamental transformations, one which allows the data flow to be changed and one which allows the execution plan to be modified:

Movable components. To optimise the data flow it is necessary to move a computation to the location of its data source. Hence, we demand that a component can be moved among nodes on the Grid. It is essential that components can move at runtime (probably repeatedly). Since components are (by definition) units of deployment, mobility of components is trivially given at deployment or load time of the application, i.e., before the application starts. Our requirement is that a component which is already executing and has been interacting with other components (and changed its internal state, for example) can be moved to a different location.

Inter-component restructuring. To change the execution plan (i.e., to replace the computation to be performed by an equivalent, but more efficient one), it is necessary for the runtime system to analyse the computation, i.e., the components and their parts, construct new components from the parts of the given components and maybe add some synthesized code to glue the parts together. This should enable a more efficient execution. In a data-centric application, restructuring may be essential to expose queries to the data sources which can be cached (i.e., a cache stash component is added) or satisfied from a cache (i.e., a cache lookup component replaces data access and filter components).

Moving components around poses the problem of suspending the execution of a component on one machine, moving the component to a different machine with a potentially different hardware and/or software environment while retaining the components state, and continuing the execution on the target machine. Together with the requirement for inter-component restructuring, this suggests the use of

bytecode (instead of native code) and probably aspect oriented techniques to achieve the restructuring.

Both actions (moving components and restructuring) occur during the runtime of an application and can be performed repeatedly. Therefore, identifying the right point in time at which to perform one or both of the actions is an important problem to solve. We propose to make this a duty of the Grid runtime system, which controls the execution of every component. As mentioned above, some component models, e.g., the Grid.it model, assign an important role to the manager of a component. In our view, the manager must be a part of the runtime system. How big the manager's influence on the runtime system's decisions regarding moving and restructuring components should be, is an unanswered question at the moment. But we suspect that managers should be written in a domain-specific language which is tailored to the job of component managers, such that the runtime system can inspect the manager and, more importantly, construct managers for newly created components after restructuring.

3.3 Metadata for Grid Components

Metadata has several uses in the context of Grid computing and component frameworks. Here, we focus on the role of metadata concerning the runtime optimisation of the execution of Grid applications. We do not discuss the use of metadata for service description, service discovery, data provenance, etc. which is the topic of the so-called Semantic Grid (see, e.g., [SBC+03]).

CoreGRID Work Package 3 has selected the Fractal Component Model as the reference model for discussing the required features of its GCM. As outlined in the introduction, the extensibility of Fractal facilitates the addition of new features in the future, which we consider important also for the GCM. In our view, the most important feature of the GCM will be automatic processing of the metadata, with which components are annotated, to drive the transformations described in Section 3.2. Therefore, an important feature of the GCM is the metadata which can be added to the components. Fractal already offers some metadata (provided that the implementation conforms to a sufficiently high level of the Fractal specification):

- a distinction between client and server interfaces, and between internal and external interfaces,
- interface and attribute names,
- a component type system to express the contingency of an interface (mandatory or optional) and the cardinality of an interface (singleton or collection).

For a Grid component model, especially one which offers the restructuring and adaptation capabilities we outline here, some more metadata is essential.

Cost functions. To govern the restructuring of a component composition, the runtime system obviously needs information to guide its decision on what to restructure and to what. Therefore, we propose to annotate components with cost functions for the computing power needed to perform a computation (in

dependence of a suitable measure of the inputs), bandwidth needed to fetch inputs and store outputs, latency generated by a component, etc.

High-level descriptions. In order to restructure a network of components by fusing, splitting and regrouping (the parts of) components, the runtime system needs to know about the function (i.e., the semantics) of the components. Since restructuring and optimising is more powerful and generally easier at higher levels of abstraction, the components must be annotated with descriptions of their function in more abstract terms than bytecode, for example. A high-level description could be given in terms of lambda expressions, domain-specific languages, polyhedral descriptions of loop nests, etc.

Since components created during transformation and restructuring shall be again annotated with metadata (so that they can participate in further restructuring), the metadata supplied by the original components must allow the derivation of the metadata for the transformed components. For example, cost functions should not be supplied as an opaque piece of code only, but as a structured data item which can be manipulated by the runtime system.

To accomplish the interoperability of components written by different authors and to make components compatible with different implementations of the runtime system, we consider one prerequisite essential: standardising *canonical metadata*. The widespread adoption of the GCM can only be achieved if the runtime optimisation and adaptation works "out of the box," with all the components a user may want to use in his/her Grid application. We propose to define, e.g., standard metrics for the input and output data in terms of the cost functions, or standard representations for the component's function, like polyhedral descriptions and other domain-specific languages.

4 Conclusion

Some frameworks which target Grid environments already offer movable components (or objects), for example ProActive [BCM+02]. To our knowledge, a restructuring system for components which exploits metadata on different levels to perform component transformations at runtime does not exist yet. We are still in the early phases of planning such a system, so we still must demonstrate the practical applicability of our ideas for the GCM.

Acknowledgements

The discussions with the partners mentioned in Subsection 3.1 have been supported by the CoreGRID network of excellence and the DAAD ARC programme. We thank the reviewers for their useful comments.

References

[BCM+02] Françoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[BCS04] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal Component Model, 2004. http://fractal.objectweb.org/specification/.

[Gri04] Grid.it. WP8 2nd Year Deliverable: High-performance component-based programming environment. Technical report, December 2004. http://www.grid.it/.

[LBK02] Peter Liniker, Olav Beckmann, and Paul H. J. Kelly. Delayed evaluation, self-optimizing software components as a programming model. In *Euro-Par 2002*, LNCS 2400, pages 666–673. Springer Verlag, 2002.

[SBC+03] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A metadata catalog service for data intensive applications. In *ACM Supercomputing Conference (Phoenix, AZ, Nov. 2003)*, 2003.

[Szy02] Clemens Szyperski. *Component Software*. Addison-Wesley Professional, 2nd edition, 2002.

[YK03] Kwok Cheung Yeung and Paul H. J. Kelly. Optimizing Java RMI programs by communication restructuring. In D. Schmidt and M. Endler, editors, *Middleware 2003: ACM/IFIP/USENIX International Middleware Conference*, LNCS 2672, pages 324–343. Springer Verlag, 2003.

Parallel program/component adaptivity management

M. Aldinucci[1], F. André[2], J. Buisson[2], S. Campa[1], M. Coppola[1], M. Danelutto[1] and C. Zoccolo[1]

[1] UNIPI, Dept. of Computer Science, University of Pisa, Largo B. Pontecorvo 3, 56127 Pisa, Italy
[2] INRIA, IRISA / Université de Rennes 1 / INSA de Rennes, avenue du Général Leclerc, 35042 Rennes, France

Abstract. Grid computing platforms require to handle dynamic behaviour of computing resources within complex parallel applications. We introduce a formalization of adaptive behaviour that separates the abstract model of the application from the implementation design. We exemplify the abstract adaptation schema on two applications, and we show how two quite different approaches to adaptivity, the ASSIST environment and the AFPAC framework, easily map to this common schema.

1 An Abstract Schema for Adaptation

With the advent of more and more complex and dynamic distributed architectures, such as Computational Grids, growing attention has to be paid to the effects of dynamicity on running programs. Even assuming a perfect initial mapping of an application over the computing resources, choices made can be impaired by many factors: load of the used machines and network available bandwidth may vary, nodes can disappear due to network problems, user requirements may change.

To properly handle all these situations, as well as the implicitly dynamic behaviour of several algorithms, *adaptivity* management code has to be built into the parallel/distributed application. In so doing, a tradeoff must be settled between the complexity of adding dynamicity-handling code to the application and the gain in efficiency we obtain.

The need to handle adaptivity has been already addressed in several projects (AppLeS [6], GrADS [11], PCL [9], ProActive [5]). These works focus on several aspects of reconfiguration, e.g. adaptation techniques (GrADS, PCL, ProActive), strategies to decide reconfigurations (GrADS), and how to modify the application configuration to optimize the running application (AppLes, GrADS, PCL). In these projects concrete problems posed by adaptivity have been faced, but little investigation has been done on common abstractions and methodology [10].

In this work we discuss, at a very high level of abstraction, a general model of the activities we need to perform to handle adaptivity in parallel and distributed programs.

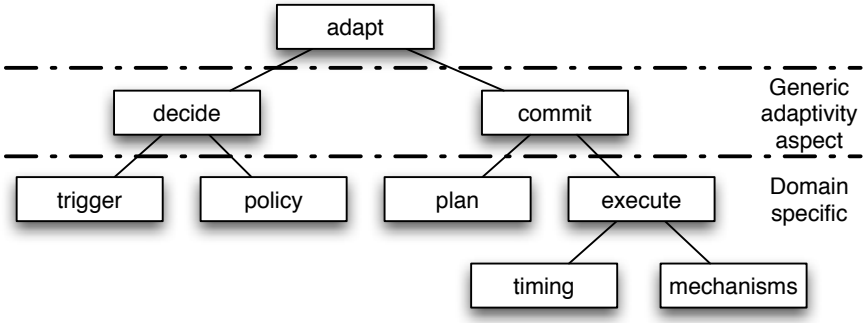


Fig. 1. Abstract schema of an adaptation manager.

Our model is abstract with respect to the implemented adaptation techniques, monitoring infrastructure and reconfiguration strategy; in this way we can uncover the common aspects that have to be addressed when developing a programming framework for reconfigurable applications, and we show that it can be applied to two concrete examples: ASSIST [4] and AFPAC [7].

The abstract model of dynamicity management we propose is shown in Fig. 1, where high-level actions rely on lower-level actions and mechanisms. The model is based on the separation of application-oriented abstractions and implementation mechanisms, and is also deliberately specified in minimal way, in order not to introduce details that may constrain possible implementations. As an example, the schema does not impose a strict time ordering among its leaves. In order to validate the proposed abstraction, we exemplify its application in two distinct, significant case studies: message-passing SPMD programs, and component-based, high-level parallel programs. In both cases, adaptive behaviour is derived by specializing the abstract model introduced here. We get significant results on the performance side, thus showing that the model maps to worthwhile and effective implementations [4].

This work has already been presented at the ParCo 2005 conference [1]. It is structured as follows. Sec. 2 introduces the abstract model. The various phases required by the general schema are detailed with examples in Sec. 3.1 and Sec. 3.2 with respect to two example applications. Sec. 4 explains how the schema is mapped in the AFPAC framework, where self-adapting code is obtained by semi automated restructuring of existing code. Sec. 5 describes how the same schema is employed in the ASSIST programming environment, exploiting explicit program structure to automatically generate autonomic dynamicity-handling code.

2 Adaptivity

The process of adapting the behaviour of a parallel/distributed application to the dynamic features of the target architecture is built of two distinct phases: a decision phase, and a commit phase, as outlined in Fig. 1. The outcome of the decide phase is an abstract adaptation strategy that the commit phase has to implement. We separate the decisions on the strategy to be used to adapt the application behaviour from the way this strategy is actually performed. The decide phase thus represents an abstraction related to the application structure and behaviour, while commit phase concerns the abstraction of the run-time

support needed to adapt. Both phases are split into different items. The decide phase is composed of:

- trigger – It is essentially an interface towards the external world, assessing the need to perform corrective actions. Triggering events can result from various monitoring activities of the platform, from the user requesting a dynamic change at run-time, or from the application itself reacting to some kind of algorithm-related load unbalance.
- policy – It is the part of the decision process where it is chosen how to deal with the triggering event. The aim of the adaptation policy is to find out what behavioural changes are needed, if any, based on the knowledge of the application structure and of its issues. Policies can also differ in the objectives they pursue, e.g. increasing performance, accuracy, fault tolerance, and thus in the triggering events they choose to react to.
  Basic examples of policy are "increase parallelism degree if the application is too slow", or "reduce parallelism to save resources". Choosing when to rebalance the load of different parts of the application by redistributing data is a more significant and less obvious policy.

In order to provide the decide phase with a policy, we must identify in the code a pattern of parallel computation, and evaluate possible strategies to improve/adapt the pattern features to the current target architecture. This will result either in specifying a user-defined policy or picking one from a library of policies for common computation patterns. Ideally, the adaptation policy should depend on the chosen pattern and not on its implementation details.

In the commit phase, the decision previously taken is implemented. In order to do that, some assessed plan of execution has to be adopted.

- plan – It states how the decision can be actually implemented, i.e. what list of steps has to be performed to come to the new configuration of the running application, and according to which control flow (total or partial order).
- execute – Once the detailed plan has been devised, the execute phase takes it in charge relying on two kinds of functionalities of the support code
  - the different mechanisms provided by the underlying target architecture, and
  - a timing functionality to activate the elementary steps in the plan, taking into account their control flow and the needed synchronizations among processes/threads in the application.

The actual adapting action depends on both the way the application has been implemented (e.g. message passing or shared memory) and the mechanisms provided by the target architecture to interact with the running application (e.g. adding and removing processes to the application, moving data between processing nodes and so on).

The general schema does not constrain the adaptation handling code to a specific form. It can either consist in library calls, or be template-generated, it can result from instrumenting the application or as a side effect of using explicit code structures/library primitives in writing the application. The approaches clearly differ in the degree of user intervention required to achieve dynamicity.

3 Examples of the abstract decomposition

In order to better explain the abstract adaptation model, we instantiate the model in two different applications, and discuss the meaning that actions and phases in the model assume.

3.1 Task farming

We exemplify the abstract adaptation schema on a task-parallel computation organized around a centralized task scheduler, continuously dispatching works to be performed to the set of available processing elements. For this kind of pattern, both a performance model and a balancing policy are well known, and several different implementation are feasible (e.g. multi-threaded on SMP machines, or processes in a cluster and/or on the Grid). At steady state, maximum efficiency is achieved when the overall service time of the set of processing elements is slightly less than the service time of the dispatcher element.

Triggers are activated, for instance, when (1) the average interarrival time of task incoming is much lower/higher than the service time of the system, (2) on explicit user request to satisfy a new performance contract/level of performance, (3) when built-in monitoring reports increased load on some of the processing elements, even before service time increases too much.

Assuming we care first for computation performance and then resource utilization, the adaptation policy would be like that in Fig. 2. Applying this policy, the decide phase will eventually determine the increase/decrease of a certain magnitude in the allocated computing power, independently of the kind of computing resources.

This decision is passed to the commit phase, where we must produce a detailed plan to implement it (finding/choosing resources, devising a mapping of application processes where appropriate).

Assuming we want to increase the parallelism degree, we will often come up with a simple plan like that in Fig. 3. The given plan is the most usual one, but some steps can be skipped depending on the implementation. For example, a multithreaded program executing on a SMP architecture does not require the code to be installed (step 2). The order may also be different, e.g. swapping steps 3 and 4. Actions listed in the plan exploit mechanisms provided by the implementation, for instance to either fork new threads, or stage and run new processes or even ask for a larger processing time share (on a multiprogrammed

- when steady state is reached, no configuration change is needed
- if the set of processing elements is slower than the dispatcher, new processing elements should be added to support the computation and reach the steady state
- if the processing elements are much faster than the dispatcher, reduce their number to increase efficiency

Fig. 2. A simple farm adaptive policy

1. find a set of available processing elements $\{P_i\}$
2. install code to be executed at the chosen $\{P_i\}$ (i.e. application code, code that interacts with the task scheduler and for dinamicity handling)
3. register with the scheduler all the $\{P_i\}$ for task dispatching
4. inform the monitoring system that new processing element have joined the execution

Fig. 3. Plan for increasing resources.

system with QoS control at the system level). The list of steps in the plan is also customized w.r.t. application implementation. As an example, whenever computing resources are homogeneous, step 1 is quite simple, while it will require a specific effort to select the best execution plan on heterogeneous resources.

Once the detailed plan has been devised, it has to be executed and its actions have to be orchestrated, choosing proper timing in order that they do not to interfere with each other and with the ongoing computation.

Abstract timing depends on the implementation of the mechanisms, and on the precedence relationship that may be given in the plan. In the given example, steps 1 and 2 can be executed in sequence, but without internal constraint on timing. Step 3 requires a form of synchronization with the scheduler to update its data, or to suspend all the computing elements, depending on actual implementation of the scheduler/worker synchronization. For the same reason, execution of step 4 also may/may not require a restart/update of the monitoring subsystem to take into account the new resources.

3.2 Fast fourier transform

The fast fourier transform can be implemented as a parallel SPMD code which distributes the matrix by lines. It alternates local computation and global matrix transposition steps. A performance model is known that predicts the optimal number of processors for such an application, depending on their power and the cost of communications. The code can thus be made adaptive, by spawning processes when new processors become available. Similarly, when some allocated processors are reclaimed by the operating system, concerned processes have to be safely terminated first. Thanks to the abstract model for dynamic adaptation, such a behavior can be easily designed.

The policy is composed of the following two statements: when the trigger notifies of available processors, and if the optimal number of processors is not overflowed, then the application decides to start new processes; when the trigger notifies that some used processors are reclaimed, some of the processes will be stopped. Given this decision, the commit phase produces a plan. The plans for the two kinds of adaptation are given on Fig. 4 and 5.

This example shows that the implementation mechanisms may depend on several aspects of the application. For example, redistributing a matrix is strongly dependent on the application and its implementation. On the other hand, prepa-

1. prepare the environment for the newly recruited processors (start daemons, stage-in files, etc.)
2. spawn processes to be executed by the new processors
3. fix connections between processes such that the new ones can communicate with the others
4. redistribute the matrix in order to balance the load amongst the whole set of processes

Fig. 4. Plan for spawning processes.

1. redistribute the matrix in such a way that the terminating processes do not hold any part of the matrix anymore
2. fix connections between processes in order to exclude the processes that are terminating
3. effectively terminate the concerned processes
4. clean everything that has been previously installed specifically for the application

Fig. 5. Plan for removing processes.

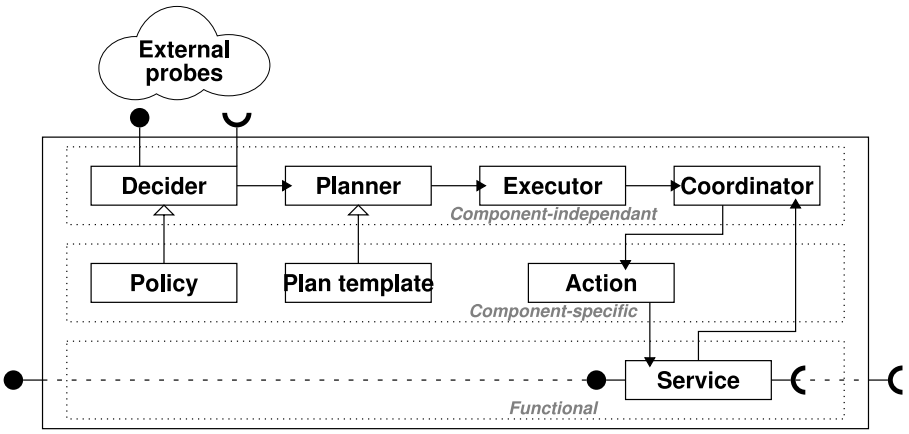


Fig. 6. AFPAC Framework.

ration of the environment may require for example starting daemons (when using LAM-MPI communications), but it is not strictly related to the application code.

The mechanisms also impose various constraints on the timing phase of the abstract model, depending on their implementation. This is the case for action 2 of the plan for spawning processes which creates the processes. For an MPI application this action can be implemented either the standard way, with the `MPI_Comm_spawn`, or in an ad-hoc way if the developer requires finer control over process creation. The former approach requires synchronization of already running processes, whereas the latter may not.

4 AFPAC: A generic framework for developers to manage adaptation

The AFPAC framework [7] focuses at present on adaptability of parallel components. Its approach consists in defining the modifications that should be applied to an existing component in order to make it able to adapt itself. Its concrete architecture (Fig.6) can be seen as a specialization of the abstract model of Fig. 1 as follows. Indeed, policy, planner and actions entities implement respectively the policy, plan and mechanisms phases of the abstract model; the timing phase of the abstract model is split over both the executor for handling the control flow and the coordinator for the synchronization with the application. AFPAC does not make appear explicitly the trigger phase as it is considered as an interface, whereas the service entity, modelling the application, has no counter-part in the abstract model. As shown in Fig.6, the decider glues the policy to the external probes in the same way that the decide phase aggregates the trigger and policy phases in the abstract model. The same kind of matching applies between the executor entity and the execute phase.

In the case of a parallel component, the service is implemented by a parallel algorithm. At runtime, it contains several execution threads distributed over a collection of processes. The AFPAC framework does not impose any constraint on communications between threads.

At the current state, the AFPAC framework includes two coordinators. The first one executes sequential actions and does not impose any synchronization

constraint with the service. It is somewhat an empty coordinator. The other coordinator aims at executing parallel actions in the context of the execution threads of the service. To do so, it requires to suspend the execution threads at a state from which such actions are allowed to be executed. Such a state is called an adaptation point. In the case of parallel codes, adaptation points must be related in order to build a global state that satisfies some consistency model. For example, in the case of SPMD codes, such a consistency model may state that all threads should execute the action from the same adaptation point. This problem has been further discussed in [7]; an algorithm has been proposed in [8] for implementing such a coordinator that looks for adaptation points in the future of the execution of the service. It is especially suitable for SPMD codes such as the ones using MPI (e.g. the fast fourier transform example given in Sec.3.2).

The AFPAC framework gives full control over dynamic adaptation to the developer. Consequently, the developer is responsible for designing and implementing the policy, plan template and action entities. In the same way, he/she has to place manually adaptation points within the source code of the service as additional statements. Nevertheless, extra preparation of the component (such as generation of annotations required by the coordinator) is done automatically thanks to aspect-oriented programming. Thanks to this semi-automated modification and to the separation of concerns, AFPAC can be used to make adaptable existing legacy codes at a low development cost.

5 ASSIST: Managing dynamicity using language and compilation approaches

ASSIST applications are described by means of a coordination language, which can express arbitrary graphs of (possibly) parallel modules, interconnected by typed streams of data. A parallel module (*parmod*) coordinates a set of concurrent activities which are performed by *Virtual Processes* (VPs). VPs execute a set of sequential activities on their input data and internal state, activities which are selected on item arrival from the input streams. The sequential functions can be programmed using standard sequential languages (C, C++, Fortran).

Overall, a *parmod* may behave in a data-parallel (e.g. SPMD/for-all/apply-to-all) or task-parallel way (e.g. farm, pipeline), and it can nondeterministically accept from one or more input streams a number of input items, which may be decomposed in parts and used as function parameters to activate VPs. A parmod may also exploit a distributed shared state, which survives between VP activations related to different stream items. More details on the ASSIST environment can be found in [12, 3].

An ASSIST module (or a graph of modules) can be declared as a component, which is characterized by provide and use ports (both one-way and RPC-like), and by *Non-Functional* ports. Among the non-functional interfaces there are those related to QoS control.

At any moment during an ASSIST application run, components can be assigned a new QoS contract, e.g. specifying a performance requirement. In order to fulfill the contracts, the component framework continuously adapts component configurations, in terms of parallelism degree, and process mapping [4]. The adaptation mechanism relies on automatic user code instrumentation, and on a hierarchy of Application Managers [2].

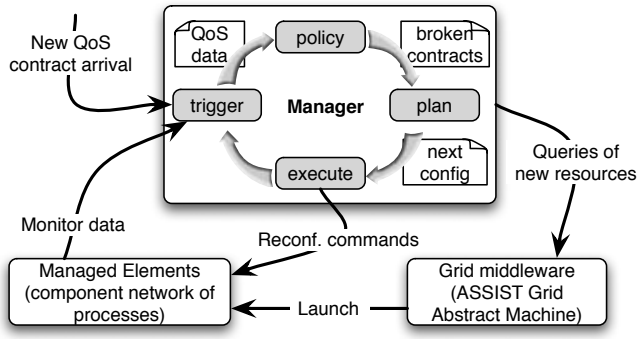


Fig. 7. ASSIST framework.

Each component has a Component Adaptation Manager (CAM) entity coordinating its adaptation. An Application Manager (AM), possibly distributed, enforces QoS of the application in the whole, by coordinating and leveraging on CAMs. As sketched in Fig. 7, ASSIST implements the abstract adaptation schema by organizing its leaves, left to right (compare with Fig. 1) in an autonomic control loop. CAM managed entities are processes within a component, while the AM applies the abstract model to application components. In the following we describe the CAM case.

The trigger functionality has to (1) collect a stream of monitoring data from the running program, as a feedback to the autonomic behaviour of AMs, and (2) to react to QoS contract changes when they trigger the need for adaptation.

The policy phase in Fig. 7 evaluates a component performance model over the monitoring data, to find out the amount/allocation of resources that can match the assigned QoS contract. In the case the QoS contract is broken, the decide phase will set a target for the commit phase, e.g. the additional amount of required computing power. The ASSIST compiler synthesizes the performance model from static information on the parallel pattern exploited by the component. Application programmers can also override standard performance models with custom ones.

The plan phase in Fig. 7 reconveys component performance within its contractually specified values by exploiting the set of actions available as mechanisms. Plan templates are instantiated as partially ordered sets of actions, which are performed according to the schedule provided by timing. ASSIST implements two layers of adaptation mechanisms: parallelism degree management (add or remove resource to/from computation), and computation (VP) remapping, with associated data migration and global state consolidation.

The timing functionality, not shown in Fig. 7, involves a distributed agreement among a set of VPs on the point where the reconfiguration must happen. In ASSIST the migration process can be performed in so-called *reconf-safe* points [4], i.e. points in the application code where the distributed computation and state are known to be consistent, and can be efficiently synchronized. Placement and use of reconf-safe points are automated, so that different mechanisms avail-

able to the execute phase (reconfiguration commands in Fig.7) automatically get the appropriate kind of synchronization.

The execute functionality thus exploits support code built within the VPs, and coordinates it with services provided by the component framework to interface to Grid middleware (e.g. for resource recruiting).

Observe that all the code needed to perform the timing and execute phases is automatically generated by the ASSIST compiler, that instruments the application code in a fully transparent manner for the application developer. ASSIST reconf-safe points are designed to exploit synchronization points already needed to ensure the correctness of the parallel application code. Moreover, the ASSIST high-level structured nature enables the compiler to automatically select the optimal implementation of mechanisms for each application and reconf-safe point. For instance, no state migration code is inserted for stateless computations, and depending on the parallelism pattern (e.g. stream versus data parallel), VPs involved in the synchronisation can be a subset of those within the component being reconfigured.

In this way ASSIST adaptive components run with no overhead with respect to non-adaptive versions of the same code, when no configuration change is performed [4].

6 Conclusions

We have described a general model to provide adaptive behaviour in Grid-oriented component-based applications. The general schema we have shown is independent of implementation choices, such as the responsibility for inserting the adaptatation code (either left to the programmer, as it happens in the AF-PAC framework, or performed by exploiting knowledge of the high level program structure, as it happens in the ASSIST context). The model also encompasses user-driven as well as autonomic adaptation.

The abstract model helps in separating application and run-time programming concerns of adaptation, exposing adaptive behaviour as an aspect of application programming, formalizing the concerns to be addressed, and encouraging an abstract view of the run-time mechanisms for dynamic reconfiguration.

This formalization gives the basis for defining a methodology. The given case studies provide with valuable clues about how to solve different concerns, and how to identify common parts of the adaptation that can be generalized in support frameworks. The model can be thus also usefully applied within other programming frameworks, like GrADS, which do not enforce a strong separation of adaptivity issues into design and implementation.

We expect that such a methodology will lead to more portable and understandable adaptive applications and components, and it will also promote layered software architectures for adaptation, simplifying implementation of both the programming framework and the applications.

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

2002-004265), and it was partially supported by the Italian MIUR FIRB project *Grid.it* (n. RBNE01KNFP) on High-performance Grid platforms and tools.

References

1. M. Aldinucci, F. André, J. Buisson, S. Campa, M. Coppola, M. Danelutto, and C. Zoccolo. Parallel program/component adaptivity management. In *ParCo 2005*, 2005. to appear.
2. M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for high performance Grid programming in Grid.it. In V. Getov and T. Kielmann, editors, *Proc. of the Workshop on Component Models and Systems for Grid Applications (June 2004, Saint Malo France)*. Springer, January 2005.
3. M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo. ASSIST as a research framework for high-performance Grid programming environments. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software environments and Tools*. Springer, 2005. (to appear, draft available as TR-04-09, Dept. of Computer Science, University of Pisa, Italy, Feb. 2004).
4. M. Aldinucci, A. Petrocelli, A. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of Grid-aware applications in ASSIST. In Jos C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference, Lisbon, Portugal, August 30 - September 2, 2005. Proceedings*, volume 3648 of *LNCS*, pages 711–781. Springer-Verlag, August 2005.
5. F. Baude, D. Caromel, and M. Morel. On hierarchical, parallel and distributed components for Grid programming. In V. Getov and T. Kielmann, editors, *Workshop on component Models and Systems for Grid Applications*, ICS '04, Saint-Malo, France, June 2004.
6. F. D. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proc. of the 1996 ACM/IEEE Conf. on Supercomputing (CDROM)*, page 39, 1996.
7. J. Buisson, F. André, and J.-L. Pazat. Dynamic adaptation for grid computing. In P.M.A. Sloot, A.G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005 (European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers)*, volume 3470 of *LNCS*, pages 538–547, Amsterdam, June 2005. Springer-Verlag.
8. J. Buisson, F. André, and J.-L. Pazat. Enforcing consistency during the adaptation of a parallel component. In *The 4th Int.l Symposium on Parallel and Distributed Computing*, July 2005.
9. B. Ensink, J. Stanley, and V. Adve. Program control language: a programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.
10. M. McIlhagga, A. Light, and I. Wakeman. Towards a design methodology for adaptive applications. In *Mobile Computing and Networking*, pages 133–144, May 1998.
11. S. Vadhiyar and J. Dongarra. Self adaptability in grid computing. *International Journal Computation and Currency: Practice and Experience*, 2005. To appear.
12. M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, December 2002.

GRID superscalar and SAGA: forming a high-level and platform-independent Grid programming environment

Raül Sirvent[1], Andre Merzky[2], Rosa M. Badia[1], and Thilo Kielmann[2]

[1] Barcelona Supercomputing Center and UPC, Barcelona, Spain
{rosa.m.badia|raul.sirvent}@bsc.es
[2] Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{merzky|kielmann}@cs.vu.nl

Abstract. The Simple API for Grid Applications (SAGA), as currently standardized within GGF, aims to provide a simple yet powerful Grid API; its implementations shielding applications from the intricacies of existing and future Grid middleware. The GRID superscalar is a programming environment in which Grid-unaware task flow applications can be written, for execution on Grids. As such, GRID superscalar can be seen as a client application to SAGA. In this paper, we discuss how SAGA can help implementing GRID superscalar's runtime system, together forming a high-level and platform-independent programming environment for the Grid.

1 Introduction

Core Grid technologies are rapidly maturing, but there remains a shortage of real Grid applications. One important reason is the lack of a simple and high-level application programming toolkit, bridging the gap between existing Grid middleware and application-level needs.

Experience shows that both Grid-aware and Grid-unaware aplications exist for their respective purposes, however having different API requirements. Typical Grid-aware applications can be taylored to the Grid, so achieving a specific use of the services provided, for example, explicitly use remote resources like data repositories. The main drawback of Grid-aware applications is the bigger effort required from the user to develop the application. Grid-unaware applications are easier from users point of view, because they don't have to care about the details in the underlying Grid infrastructure. They simply need to complete their task, irrespective of the machinery involved. The drawback in this case is that you cannot use more advanced services offered in the Grid, because the Grid-unaware infrastructure uses the services by you. From this we can see that the two groups of users, Grid-aware and Grid-unaware, are completely opposite. The first want to extract all the functionality available to the Grid, while the last want to run their work in the Grid with a minimum effort.

For both categories of applications, targeted programming environments exist. In this paper, we anticipate an integrated approach, as sketched in Fig. 1.

In particular, we investigate the integration of GRID superscalar [4] and of SAGA [10], aiming at merging, kind of, "the best of both worlds." An important part of this exercise is to implement GRID superscalar's runtime system using the SAGA interface only, allowing for a both high-level and platform-independent Grid programming environment.

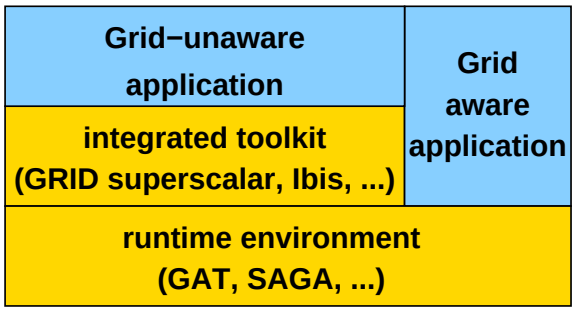


Fig. 1. Integrated runtime platform

The remainder of this paper is organized as follows. Sections 2 and 3 will sketch the SAGA API and GRID superscalar, respectively. In Section 4, our anticipated integration of both will be presented. Section 5 will briefly sketch related work, and Section 6 concludes.

2 The Simple API for Grid Applications (SAGA)

The SAGA API specification as it is currently being defined by GGF focuses on four components widely cited in Grid computing use cases. These components and their respective, most important method calls are:

1. **Files:** management and access to files on remote storage resources:
   - `directory.{cd, list, copy, link, move, delete, exists, create, open, open_dir, ...}`
   - `file.{read, write, seek}`

2. **Logical Files:** management and access to replica systems for large distributed data Grids:
   - `logical_directory.{cd, list, copy, link, move, delete, exists, create, open, open_dir, ...}`
   - `logical_file.{add_location, remove_location, list_locations, replicate}`

3. **Jobs:** starting and controlling jobs running on remote Grid resources:
   - `job_description.{set_attribute, get_attribute, ...}`
   - `job_server.{list_jobs, run_job, submit_job}`
   - `job.{suspend, resume, hold, release, checkpoint, migrate, ...}`

   - `job.{get_id, get_status, get_exit_status, ...}`

4. **Streams:** exchange of application specific data via a secured, high throughput streaming channel:
   - `stream_server.{wait_for_connection}`
   - `stream.{connect, read, write, status, wait}`
   - `multiplexer.{watch, unwatch, wait}`

This list of components and methods is very concise on purpose, but it allows the implementation of a large number of simple Grid use cases. As the SAGA API is abstracting and simplifying the paradigms provided by the various Grid middleware incarnations, SAGA implementation can provide a very simple and stable environment, protecting the user from the evolution of Grid middleware.

Along with the mentioned API components, the API draft also defines a common look-and-feel for the components, and for future extensions. Additionally, the so called API core encompasses:

- session handling,
- encapsulation of security information,
- error handling,
- a generic task model.

In particular the task model is important to many Grid applications, as it allows both the asynchronous execution of potentially slow, remote operations, and the ability to react on status changes for pending remote operations. Fig. 2 shows an example of an asynchrounous file read operation using the task model. The SAGA API will likely be extended in the future, with additional paradigms such as GridRPC, access to information services, monitoring and steering.

3 The GRID superscalar programming environment

In contrast to SAGA, GRID superscalar [4] tries to hide the Grid from the user, in such a way that writing an application for a computational Grid may be as easy as writing a sequential application. From the source code provided by the user, and taking into account the functions in the code selected to be Grid-enabled, the run-time system generates a directed, acyclic graph (DAG) of these functions where the dependencies are defined via data files. From that graph, the run time system executes all those functions for which data dependencies become resolved, therefore achieving automatic functional parallelism for the original sequential program. The GRID superscalar framework is mainly composed of the programming interface, the deployment center and the run-time system. It acts as an assembly language for the Grid, as it allows to use different programming languages on top of it.

The programming interface offers a very small set of primitives. These primitives must be used in the main program of the application for different purposes.

Fig. 2. Code Snippet: *asynchronous file reading with the SAGA task model*

```
{
  // synchronous operations
  saga::file   f (url);
  string out = f.read (100);
  std::cout << out << std::endl;
}
{
  // asynchronous operations
  saga::file f (url);
  saga::file::task_factory ftf = f.get_task_factory ();

  string out;
  saga::task t = ftf.read (100, out);

  t.run ();

  while ( saga::task::Done != t.get_status () )
    { sleep (1); }

  std::cout << out << std::endl;
}
```

GS_On() and GS_Off() primitives are provided for initialization and finalization of the run-time. GS_Open(), GS_Close(), GS_FOpen() and GS_FClose() for handling files. The GS_Barrier() function has been defined to allow the programmers to wait till all Grid tasks finish. Also the GS_Speculative_End() function allows an easy way to implement parameter studies. In order to specify the functions to be executed in the Grid, an IDL file must be used. For each of these functions, the type and direction of the parameters must be specified (where direction means if it is an input, output or input/output parameter.)

The deployment center is a Java-based Graphical User Interface. It implements the Grid resource and application configuration, an early failure detection, the source code file transfer to the remote machines, the generation of additional source files required for the master and the worker parts (using the gsstubgen tool), the compilation of the main program in the localhost and worker programs in the remote hosts and finally the generation of the configuration files needed for the run-time. The call sequence is presented in Fig. 3).

The run-time library is able to detect the inherent parallelism of the sequential application and performs concurrent task submission. Techniques such as file renaming, file locality, disk sharing, checkpointing or ClassAds [11] constraints specification are applied to increase the application performance, save computation time already performed or select resources in the Grid. Currently, the machine list has been enabled to be dynamic, so at execution time a user can specify new machines added to the computation, and erase machines which are no more available, as well as changing features from machines (i.e. limit of jobs, submission queue, software available, etc.) Different versions of the run-time exist in order to use different Grid middleware technologies, such as Globus 2.4 [8], Globus 4 [8], Ninf-G2 [12] and ssh/scp. Extra layers between GRID superscalar

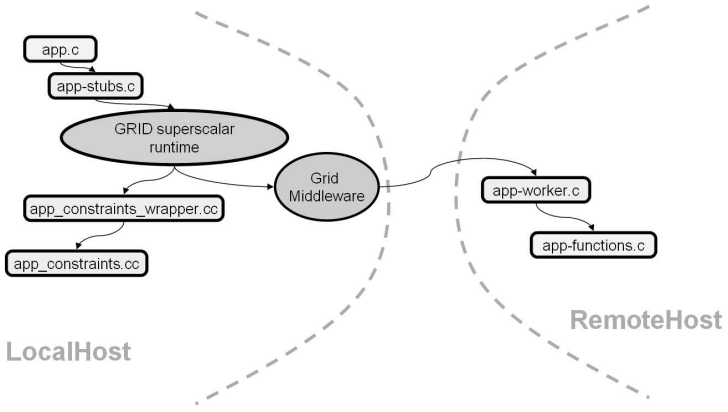


Fig. 3. GRID superscalar behavior

and the underlying Grid middleware, such as SAGA and GAT [2], could ease the step of porting GRID superscalar to new Grid middleware, which is known to be a tedious task.

4 Implementing the GRID superscalar runtime system using SAGA

As shown in Fig. 1, a SAGA layer within the GRID superscalar runtime system replaces the current, direct middleware bindings. This section will show how this replacement works, using the Globus based GRID superscalar run-time version as example. We picked central Globus code snippets as used in GRID superscalar, and compare them to the respective SAGA replacements[3].

4.1 Initialization and Session Management

The Globus module initialization is performed once per session, and does not require more than a single call for each Globus module used. SAGA initializes a default session handle, which is transparently used (see listing in Fig. 4). However, the user *can* have tighter control on that handle, if needed.

4.2 File Movement

Moving files in a Grid from A to B is probably *the* most common use case for Grid applications. Globus provides various means to perform such operations, with different complexity, performance, and behavior. One of the simplest versions is provided by the Global Access to Secondary Storage (GASS) module, as shown in Fig. 5.

One of the major problems for file movement in Grids is that the application programmer or even the end user needs to be aware of the transport protocols

[3] the code snippets are slightly simplified, e.g. do not include proper error handling.

Fig. 4. Code Snippet: *Middleware Initialization*

```
globus_module_activate   (GLOBUS_GASS_COPY_MODULE);
globus_module_activate   (GLOBUS_IO_MODULE);
globus_module_activate   (GLOBUS_GRAM_CLIENT_MODULE);
globus_module_activate   (GLOBUS_COMMON_MODULE);

globus_module_deactivate_all ();
```

```
// no initialization needed in SAGA
```

available: in the code example, the user needs to know that gsiftp is actually available for that host. Although difficult to solve in general, the SAGA API provides a more abstract notion of that transport, by allowing the 'protocol' *any*. The SAGA implementation is then selecting an available protocol for the data transfer.

Fig. 5. Code Snippet: *Copying files (from a remote host to the master)*

```
globus_gass_copy_handle_init (&handle, GLOBUS_NULL);
globus_io_file_open ("/home/workingdir/file.txt",
       GLOBUS_IO_FILE_WRONLY | GLOBUS_IO_FILE_CREAT
                             | GLOBUS_IO_FILE_TRUNC,
       GLOBUS_IO_FILE_IRUSR  | GLOBUS_IO_FILE_IWUSR
                             | GLOBUS_IO_FILE_IRGRP,
       GLOBUS_NULL, &fileHandle);
globus_gass_copy_url_to_handle (&handle,
       "gsiftp://hostname/path/file.txt",
       GLOBUS_NULL, &fileHandle);
globus_io_close (&fileHandle);
```

```
saga::directory dir ("/home/workingdir/");
dir.copy ("file.txt", "any://hostname/path/file.txt");
```

4.3 Remote Job Submission

The simplest way to submit a remote job via Globus is to provide a RSL description of the job to the GRAM module (see Fig. 6). This module will forward the job submission request to the Globus gatekeeper on the remote host, which will then run the job. A job handle is returned for later reference, e.g., for checking the job's status.

The SAGA API has a similar notion of a job description. The keywords used for describing the job are compatible to those specified in GGF's JSDL [3] and DRMAA [5] specifications. As indicated in Section 2, the returned job object also allows performing a number of operations on the job, among which is checking the current job status.

Fig. 6. Code Snippet: *Job submission*

```
// The callback_contact is passed in order
// to receive notifications
sprintf (RSL, "&(executable=/path/exec)(directory=/path)"
  "(arguments=-l)(queue=short)"
  "(file_stage_in = gsiftp://host/path/file1 /path/file1)"
  "(file_stage_out= /path/file2 gsiftp://host/path/file2)"
  "(file_clean_up = /path/file3)"
  "(environment=(NAME1 val1)(NAME2 val2))");
globus_gram_client_job_request (hostname, RSL,
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE  |
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_PENDING |
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE    |
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED,
  callback_contact, &job_contac);
```

```
saga::jobdescription jd;

std::list <const char*> hosts; hosts.push_back ("host");
jd.add_vector_attribute ("SAGA_HostList", hosts);

jd.add_attribute ("SAGA_JobCmd",   "/path/exec");
jd.add_attribute ("SAGA_JobCwd",   "/path");
jd.add_attribute ("SAGA_JobArgs",  "-l");
jd.add_attribute ("SAGA_Queue",    "short");
jd.add_attribute ("SAGA_FileTransfer",
                  "/path/file1 < gsiftp://host/file1");
jd.add_attribute ("SAGA_FileTransfer",
                  "/path/file2 > gsiftp://host/file2");
jd.add_attribute ("SAGA_JobEnv", "NAME1=val1 NAME2=val2");
saga::job_server js;
saga::job        job = js.submit_job (jd);
```

4.4 Job State Notification

Via callbacks and blocking polls, Globus provides a very convenient way for GRID superscalar to wait for the completion of submitted Jobs (see Fig. 7). The SAGA API does currently not provide similar mechanisms. However, a monitoring extension to SAGA is in the planning stage, and will similarily allow to add callbacks for changes in the job status metric. SAGA also does not yet provide blocking synchronization points.

4.5 Job Manipulation

The manipulation of jobs, such as `cancel, kill` or `signal`, is solved similar in Globus and SAGA (see Fig. 8). However, the SAGA API includes more complex operations such as `migrate`, which are more difficult to implement in the less abstract Globus API's.

4.6 Job State

The set Globus and SAGA job states are somewhat different. We have focused on the equivalence of states needed from GRID superscalar's point of view,

Fig. 7. Code Snippet: *Job state notifications*

```
// TaskEnds treats the state change
globus_gram_client_callback_allow (TaskEnds, NULL,
                                   &callback_contact);
// Blocked waiting for notifications
globus_poll_blocking ();
```

```
saga::metric m = job.find_metrics ("Status");
             m.add_callback (TaskEnds);
while ( ! all_jobs_done ) { sleep (1); }
```

Fig. 8. Code Snippet: *Job cancellation*

```
globus_gram_client_job_cancel (job_contact);
```

```
job.cancel ();
```

thus some states can have a slight semantic difference between them, but they accomplish the minimum semantic needed when working with GRID superscalar. The SAGA API state diagram is very complete, so we can see the Globus states included in the SAGA states. A mapping between both is straight forward: Globus states Pending, Active, Failed and Done are known as Queued, Running, DoneFail and DoneOK in SAGA, respectively.

5 Related work

The SAGA developments are mostly driven by the Grid Application Toolkit (GAT) [2] and the Java CoG kit [15]. While the latter aims at providing a simple API to the Globus toolkit [8], the GAT aims both at a simple API and at middleware-independent implementations. The SAGA API, once standardized by GGF, will allow implementations that are either independent of or integrated into Grid middleware packages.

While GAT and SAGA provide simple API's for Grid-aware applications, higher-level toolkits like GRID superscalar aim at supporting Grid-unaware applications, thus completely hiding the underlying Grid infrastructure. Systems like Ibis [14], Assist [1], and ProActive [6] have similar aims, each providing their own flavors of high-level API's. We believe, however, that GRID superscalar's task-flow model offers a very widely applicable and efficient programming model, which is why we combine is with SAGA to form a Grid programming environment which is both high-level and platform-independent.

Overall, an integration of GRID superscalar and SAGA fits into the bigger picture of building an integrated Grid platform, as envisioned within the Core-

GRID community [7, 9, 13]. In such an integrated platform, both Grid-aware and unaware runtime interfaces will be embedded in a component system with enriched functionality like information providers, resource brokers, and application steering interfaces.

6 Conclusions

The Simple API for Grid Applications (SAGA), as currently standardized within GGF, aims to provide a simple yet powerful Grid API; its implementations shielding applications from the intricacies of existing and future Grid middleware. The GRID superscalar is a programming environment in which Grid-unaware task flow applications can be written, for execution on Grids. As such, GRID superscalar can be seen as a client application to SAGA.

In this paper, we have discussed how SAGA can help implementing GRID superscalar's runtime system, together forming a high-level and platform independent programming environment for the Grid. We have shown that the SAGA API, although simplistic on purpose, already provides almost all functionality needed for enabling a system as powerful as GRID superscalar. The only deficiency is with supporting upcalls and asynchronous event notifications, an issue currently being worked on within GGF's SAGA group. Once completed, the main benefit that GRID superscalar will get is that SAGA will provide an interface that will allow to run unmodified across various Grid middleware systems such as various versions of Globus, Unicore, and even using ssh/scp, or purely local on individual computers. So, SAGA acts as an extra layer, that makes GRID superscalar independent from Globus implementation details. The price that GRID superscalar has to pay is indeed having to add this extra layer in order to use the Grid services, which we think is low in contrast to the benefit obtained.

Acknowledgments

This work is partially funded by the European Commission, via the Network of Excellence *CoreGRID* (contract 004265). Part of the work was carried out in the context of the Virtual Laboratory for e-Science project (*www.vl-e.nl*), supported by the Dutch Ministry of Education, Culture and Science (OC&W). It is also supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01.

The SAGA API was defined by GGF's SAGA Research Group, and in particular by the SAGA design team: Shantenu Jha, Tom Goodale, Gregor von Laszewski, Andre Merzky, Hrabri Rajic, John Shalf, and Chris Smith.

References

1. M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured iplementation of component based grid programming environments. In *Future Generation Grids*. Springer Verlag, 2005.

2. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.
3. A. Anjomshoaa, F. Brisard, R. L. Cook, D. K. Fellows, A. Ly, S. McGough, and D. Pulsipher. Job Submission Description Language (JSDL) Specification Version 1.0. Draft Recommendation, Global Grid Forum (GGF), 2005.
4. R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.
5. R. Brobst, W. Chan, F. Ferstl, J. Gardiner, J. P. Robarts, A. Haas, B. Nitzberg, H. Rajic, and J. Tollefsrud. Distributed Resource Management Application API Specification Version 1.0. GFD.022 - Proposed Recommendation, Global Grid Forum (GGF), 2004.
6. D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, September-November 1998.
`http://www-sop.inria.fr/oasis/proactive/`.
7. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems. Roadmap version 1 on Problem Solving Environments, Tools, and GRID Systems. CoreGRID deliverable D.ETS.01, 2005.
8. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.
9. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12), 2002.
10. GGF. Simple API for Grid Applications Research Group, 2004.
`http://forge.gridforum.org/projects/saga-rg/`.
11. M. Solomon. The ClassAd Language Reference Manual.
`http://www.cs.wisc.edu/condor/classad/`,.
12. Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.
13. J. Thiyagalingam, S. Isaiadis, and V. Getov. Towards Building a Generic Grid Services Platform: a component-oriented approach. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*. Springer Verlag, 2005.
14. R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: A Flexible and Efficient Java-based Grid Programming Environment. *Concurrency & Computation: Practice & Experience*, 17(7-8):1079–1107, June-July 2005.
15. G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):643–662, 2001.
`http://www.cogkits.org`.

Skeleton Parallel Programming and Parallel Objects

Marcelo Pasin[1], Pierre Kuonen[2], Marco Danelutto[3], and Marco Aldinucci[4]

[1] CoreGRID fellow, on leave from Universidade Federal de Santa Maria, Brazil
[2] Haute Ecole Specialisée de Suisse Occidentale (HES-SO/EIA-FR), Fribourg, Switzerland
[3] Departimento d'Informatica, Univesità di Pisa, Italy
[4] Istituto di Scienza e Tecnologia dell'Informazione (CNR/ISTI), Pisa, Italy

Abstract. We describe here the ongoing work aimed at integrating the POP-C++ parallel object programming environment with the ASSIST component based parallel programming environment. Both these programming environments are shortly outlined, first. Then several possibilities of integration are considered. For each one of these integration opportunities, the advantages and synergies that can be possibly achieved are outlined and discussed. Eventually, the current status of integration of the two environments is discussed, along with the expected results and fallouts on the two programming environments.

1 Introduction

This is a prospective paper on the integration of ASSIST and POP-C++ tools for parallel programming. POP-C++ is a C++ extension for parallel programming, offering parallel objects with asynchronous method calls. Section 2 describes POP-C++. ASSIST is a skeleton parallel programming system that ofers a structured framework for developing parallel applications starting from sequential components. ASSIST is described in section 3 and some of its components, namely GEA and ADHOC, are described in sections 3.1 and 3.2 respectively.

This paper describes some initial ideas of cooperative work on integrating parts of ASSIST and POP-C++, in order to obtain a broader and better range of parallel programming tools. It has been clearly identified that the distributed resource discovery and matching, as well as the distributed object deployment found in ASSIST could be used also by POP-C++. An open question, and an interesting research problem, is whether POP-C++ could be used inside skeleton components for ASSIST. Section 4 is consacrated to these discussions.

2 Parallel Object-Oriented Programming

It is a very common sense in software engineering today that object-oriented programming and its abstractions improves software development. Besides that, the own nature of objects incorporate many possibilities of program parallelism. Several objects can act concurrently and independently from each other, and several operations in the same object can be concurrently carried out. For these reasons, a parallel object seems to be a very general and straightforward model for parallel programming.

POP stands for Parallel Object Programming, a programming model in which parallel objects are generalizations of traditional sequential objects. POP-C++ is an extension of C++ that implements the POP model, integrating distributed objects, several remote method invocations semantics, and resource requirements. The extension is kept as close as possible to C++ so that programmers can easily learn POP-C++ and existing C++ libraries can be parallelized with little effort. It results in an object-oriented system for developing high-performance computing applications for the grid [13].

POP-C++ incorporates a runtime system in order to execute applications on different distributed computing tools [10,17]. This runtime system has a modular object-oriented service structure. All services are instantiated inside each application and can be combined to perform specific tasks using different system services. This design can be used to glue current and future distributed programming toolkits together to create a broader environment for executing high performance computing applications.

Parallel objects have all the properties of traditional objects, added to distributed resource-driven creation and asynchronous invocation. Each object cration has the ability to specify its requirements, making possible transparent optimized resource alocation. Each object is allocated on a separate address space, but references to an object are shareable, allowing for remote invocation. Shared objects with encapsulated data allow programmers to implement global data sharing in distributed environments. In order to share parallel objects, POP-C++ can arbitrarily pass them from one place to another as arguments of method invocations. The runtime system is responsible for managing parallel object references.

Parallel objects support any mixture of synchronous, asynchronous, exclusive or concurrent method invocations. Without an invocation, a parallel object lies in an inactive state, only being activated a method invocation request. Syntactically, method invocations on POP objects are identical to those on traditional sequential objects. However, each method has its own invocation semantics, specified by the programmer. These semantics define different behaviours at both sides of the parallel object, called the interface and the object-side semantics.

The interface semantics affect the caller of a method invocation, which can be either synchronous or asynchronous. With **synchronous** invocation, the caller blocks until the execution of the requested method on the object side is finished. This corresponds to the traditional (remote) method invocations. **Asynchronous** invocations, on the contrary, return immediately after sending the request to the remote object. Asynchronous invocation is important to exploit the parallelism because it enables to overlap computations and communications. No computing result is available when the invocation returns to the caller, so, under the current model, it cannot produce results.

The object-side semantics rule the execution of methods inside each object. A method can be of one of three types: concurrent, sequential, or mutex. Invocations to **concurrent** methods are executed concurrently if no mutex invocation is currently running. The concurrent invocation is important to achieve the parallelism inside each parallel object and to improve overlapping between computation and communication.

Using **sequential** invocation, methods are executed in mutual exclusion, following the requests' arrival order. Several simultaneous sequential methods invocations are served sequentially (see Fig. 1). Concurrent methods that have been previously started

can still continue their normal execution. This guarantees the serializable consistency of all sequential invocations in the same object.

Invocations to **mutex** methods are executed in complete exclusion with all other methods of the same object. A request is executed only if no other invocation are running. Otherwise, the current method will be blocked until all invocations (including concurrent ones) are terminated (see Fig. 1). Mutex invocations are important to synchronize concurrencies and to assure the correctness of shared data state inside the parallel object.

Fig. 1. Exampe of different invocation requests

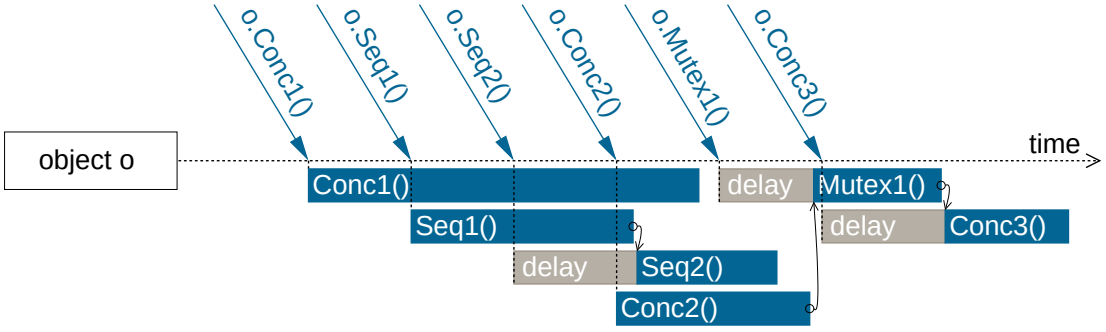


Figure 1 illustrates different invocation semantics. Sequential invocation `Seq1()` is served immediately, running concurrently with `Conc1()`. Although the sequential invocation `Seq2()` arrives before the concurrent invocation `Conc2()`, it is delayed due to the current execution of `Seq1()` (no order between concurent and sequential invocations). When the mutex invocation `Mutex1()` arrives, it has to wait for other running methods to finish. During this waiting, it also blocks other invocation requests arriving later, as `Conc3()`, until the mutex invocation request completes its execution.

Prior to allocate a new object it is necessary to select an adequate placeholder. Similarly, when an object is no longer in use, it must be destroyed to release the resources it is occupying. POP-C++ provides in the runtime system automatic placeholder selection, object allocation, and object destruction. This automatic features result in a dynamic usage of computational resources and gives to the applications the ability to adapt to changes in both the environment and application behaviour.

Resource requirements can be expressed by the quality of service that components require from the environment. POP-C++ integrates the requirements into parallel objects under the form of resource descriptions. Each parallel object constuctor is associated with an **object description** that depicts the characteristics of the resources needed to create the object. The resource requirements in object descriptions are expressed in terms of resource (host) name, computing power, amount of memory, expected communication bandwidth and latency.

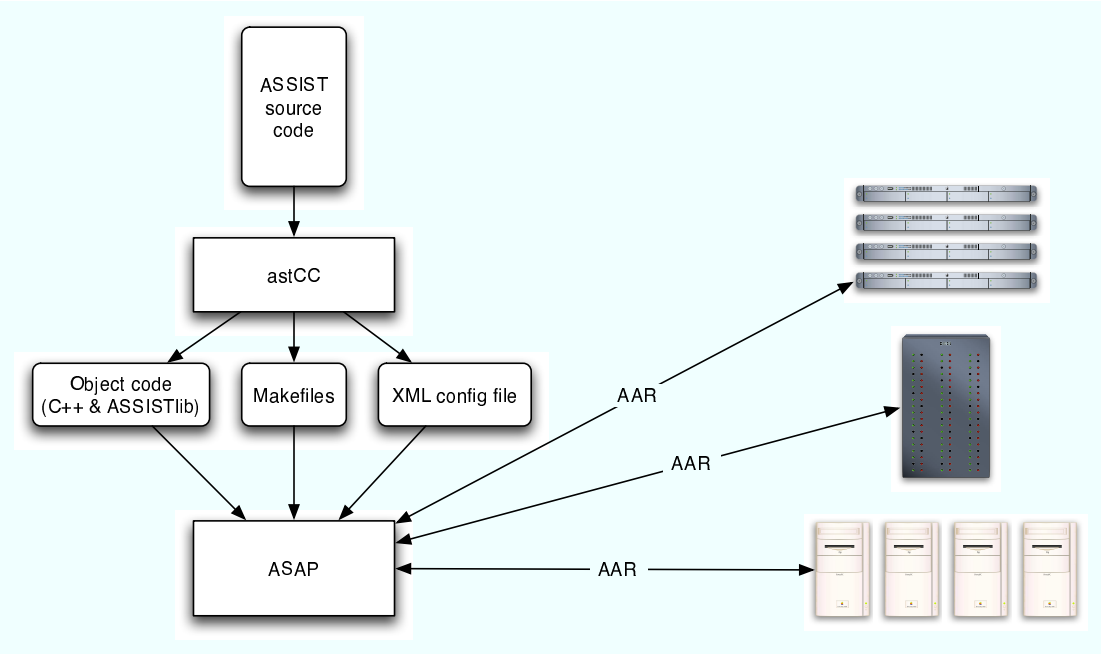
3 Structured parallel programming with ASSIST

The development of efficient parallel programs is especially difficult with large-scale heterogeneous and distributed computing platforms as the grid. Previous research on that subject exploited **skeletons** as a parallel coordination layer of functional modules,

made of conventional sequential code [3]. This model allows to relieve programmer from many concerns of classical, non structured parallel programming frameworks. As an example, scheduling, mapping, load balancing and data sharing are all managed by either the compile tools or the runtime systems of structured parallel programming frameworks. In addition to that, due to the exposition by the programmer in the program source code of the structure of parallelism exploitation, several optimizations can be efficently implemented at either compiler or runtime level. That is not applicable in case the parallellism exploitation pattern is not available or it has to be mined from source code.

ASSIST is a parallel programming environment providing a skeleton based coordination language. A compiler and a set of runtime tools allow ASSIST programs to be run on clusters, networks of workstations and grids. Several optimizations are performed that allow to achieve high efficiency in the execution of ASSIST programs [15,1]. Its programming environment was recently extended to support GRID.it components [2]. They can as well be used to interact with non GRID.it components, in particular with CORBA components and with Web Services. ASSIST GRID.it components are supplied with autonomic managers [4] that adapt the component execution to dynamic changes in the grid features (node or link faults, different load levels, etc.).

Fig. 2. ASSIST structure



The structure of the ASSIST programming environment is outlined in Fig. 2. Source code is processed by the ASSIST compiler, producing C++ code, makefiles to be used to produce the actual object code for several different architectures, and an XML configuration file that represent the descriptor of the parallel application. To run the program, this XML file is processed by the GEA tool (see section 3.1), taking care of all the activities needed to stage the code at remote nodes, starting auxiliary runtime processes, starting application code and gathering results back to the node where the program has been launched. Some parts of the system processes launched with the application code of an ASSIST program are related to ADHOC ASSIST subsystem. ADHOC is basi-

cally a shared data resource that is used to support both data repository and stream communication.

3.1 Grid Application Deployment

The GEA ASSIST tool is a parallel process launcher targeting two distinct architectures: Globus grids and POSIX/TCP workstation networks and clusters supporting SSH access. GEA takes as an input an XML file generated by the ASSIST compiler out of the ASSIST source code and an AAR file (Assist ARchive file), hosting the code and the libraries needed to deploy the ASSIST program on a remote node.

The XML file is parsed by GEA, then a suitable number of computing resources (nodes) are recruited to host the application processes. In case of Globus, resource recruitement is performed interacting with standard MDS services. In case of POSIX/TCP SSH architectures, POSIX commands are used in conjunction with SSH. The application code is deployed to the selected remote nodes, by transferring to them the proper AAR files, then the archive files are uncompressed and unpacked. The object code and libraries are then trasferred to the proper places in the local filesystems.

The necessary support processes to run the applications are also started at the selected nodes. In particular, the HOC processes used to implement the data flow streams interconnecting ASSIST processes are started in this step. Eventually, the processes derived from and implementing the user code are run. They perform user defined code upon the data received from the HOC implemented data flow streams, which eventually deliver results again on the HOC channels.

All these steps can be performed exploiting two different kinds of technologies: Globus and SSH. With **Globus** (toolkit 2.4, currently moving to toolkit 4), the resource lookup is performed exploiting MDS facilities, data and code (AAR) staging is performed via GlobusFTP and processes are run remotely exploiting Globus remote commanding facilities. **SSH** is a standard mechanism to run remote commands and to transfer files, natively available by classical POSIX operating systems and supported, non natively, also by Windows. Code and data staging is performed using `scp`, remote processes are started via `ssh` and resources are looked for by inspecting a file or by a special lookup process testing access to the machines on the local network.

The whole process not only supports the user code launch, but also the management of all the runtime processes needed to monitor ASSIST program performance and possibly to force the program to terminate, or even to adapt (e.g. varying its parallelism degree) to changements in the grid architecture features and/or in the perfromance contracts issued by the users.

ASSIST GEA is currently being engineered by separating the code performing actions from the code planning the application deployment. The main GEA code implements a **plugin manager** built on top of the COG toolkit [16]. The plugin manager basically is able to load and run a module configured according to the XML file tags. The plugin, in turn, is able to perform all the actions needed to deploy and run a code developed with a particular environment. As an example, the ASSIST plugin works as described above, by first stagin and running the ADHOC code, than staging and running the ASSIST user code. A CORBA/CCM [11] plugin first sets up the CORBA framework and then launches the CCM code wrapped in the ASSIST program.

3.2 Distributed Data Collections

To profit from the large processing potential of the grid, applications cannot assume the platform to be neither homogeneous, secure, reliable nor centrally managed. Also, these applications should be fed with large distributed collections of data.

ADHOC (Adaptive Distributed Herd of Object Caches), is a distributed object repository [5]. It has been conceived in the context of the ASSIST project, as a distributed persistent virtual storage. It provides the application designers with a toolkit to solve data storage problems in a grid framework. In particular, it provides building blocks to set up client-server and service-oriented infrastructures which can cope with the grid characteristics. Its underlying design principle consists in decoupling the management of computation and storage in distributed applications.

Parallel grid applications often need processing large amounts of data. Data storages for such applications are required to be fast, dynamically scalable and enough reliable to survive to some failures. Decoupling helps in providing a broad class of parallel applications with these features while achieving good performances. ADHOC creates a local virtual memory associated with every processing element. A common distributed data repository is provided by the cooperation between multiple local memories.

ADHOC implements an external repository for arbitrary length objects. Clients may access objects through different protocols, implemented within proxy libraries. Proxies may act as a simple adaptors, or exploit complex behaviors, even cooperating with other proxies (e.g. distributed agreement). An object cannot be spread across different nodes, but it can be replicated. Objects can be grouped in ordered collections of objects, which can be spread across different nodes.

Objects and collections are identified by keys. The actual data location is found at execution time through a distributed hash table. ADHOC API enables to get, put and remove objects, and it provides remote execution of objects methods. This operation is initially meant as mechanism to extend server core functionalities for specific needs, as for example lock and unlock the object for consistency management.

4 Exploiting POP-C++ and ASSIST synergies

POP-C++ and ASSIST have been designed and developed with different programming models in mind, but with a common goal: provide grid programmers with advanced tools suitable to be used to develop efficient grid applications. Some of the problems addressed and (partially) solved in the two contexts are therefore common problems. In particular, the way active entities (objects in POP-C++ and processes in ASSIST) are deployed to the grid processing nodes, the kind of support needed to efficiently share data and the way parallelism can be exploited in a single grid program component are all subject of design and implementation efforts in both these environments.

In this section, we want to address the synergies that can be exploited among POP-C++ and ASSIST. We want to consider the possibilities of integrating the POP-C++ and the ASSIST environments and, in particular, the integration possibilities that effectively improve one of the two environments exploiting the original results already achieved in the other environment. Three kind of possibilities have been explored, that seem to provide suitable improvements in either the ASSIST or the POP-C++ environments:

1. to exploit the ASSIST GEA deployment tool to deploy and manage POP-C++ programs
2. to exploit ASSIST ADHOC shared memory support to implement shared state in POP-C++ programs
3. to use POP-C++ to implement GRID.it components in the ASSIST framework

The former two cases actually improve the possibilities offered by POP-C++ by exploiting ASSIST technology. The latter case improves the possibilities offered by ASSIST to assemble complex programs out of components written accordingly to different models. Currently such components can only be written using the ASSIST coordination language or inherited from CCM or Web Services. The following sections detail these three possibilities and discuss their relative advatages/disadvantages.

4.1 Exploiting ASSIST GEA in POP-C++

POP-C++ comprises a runtime library that implements some services for launching remote processes and for resource discovery. Launching remote processes is provided by a **job manager**, which has two main functionalities: launching the parallel object and managing the resources. It allows to submit jobs with different management systems such as LSF [9], PBS [12] or even Globus [10]. It does not provide authentication services and relies on the security infrastructure of the management system used.

A distributed resource discovery is integrated in the POP-C++ runtime system. It differs from the centralized approach such as in Globus, NetSolve [8] or Condor [14]. Information about the POP-C++ resources is fully distributed and accessed on demand, configuring an adaptive peer-to-peer model. Though, this model has shown some scalability problems and it is a good candidate for a replacement.

GEA provides a comprehensive infrastuture for launching processes, integrated with functions for matching needs to resouces capabilities. The integration of POP-C++ with GEA could be done in different levels. The most straightforward would be to replace the parts of the job manager related to object loading and running and the resource discovery with calls to GEA, which would perform all launching and all resource management. In any case, POP object files would have to be packed into ASSIST application packages, which is the file format understood by GEA.

In order to assess the implications of the integration proposed here, the object creation procedure inside the job manager has to be seen more into detail. Initially, a proxy object is created, called interface. The interface evaluates the object description and calls the resource discovery service to find a suitable resource. The interface then launches an object server in the given resource. The object server now running in the resource takes care of all other tasks, as downloading and starting the executable code, setting the connection with the interface, receiving the constructor arguments and signalling the interface about the end of the creation.

The discovery service as required by the interface is not yet implemented in GEA. If implemented, GEA, should return an access point to the resource found. As GEA can be instructed to load and launch a program in a specified resource, the interface algorithm could stay as it is. On the other hand, instead of adding a discovery call to GEA, the interface algorithm could be changed. It could directly ask GEA to launch the

new object using a resource description. This is also present in GEA, but only could be used with some modification.

Requests to launch processes have some restrictions on GEA. Its currently structured model matches the structured model of ASSIST. Nodes are divided into domains. The ASSIST model dictates a fixed structure for parallel programs, which are formed by parallel modules, that are connected in a predefined way. Modules are divided into processes, which are assigned to resources when the execution starts. All resources assigned to a single parallel module must belong to the same domain. It is eventually possible to adjust on the number of processes inside of a running parallel module, but the new processes must be started in the same domain.

POP-C++ needs a completely dynamic model to run parallel objects. An object running in a domain must be able to start new objects in different domains. In order to support that, GEA has to be extended to a more flexible model. This can be done by making a process launching interface accessible from inside a domain. Also, resource discovery for new processes must take into account the resources in all domains (not only the local one). This functionalities can be added to GEA either as plugins or as a separate process, as is the case of the ADHOC server.

In most grid systems, node allocation is based on some sort of application requirements and on resource capabilities. In the context of POP-C++ (an in other similar systems, as ProActive [7], for instance), the allocation must be done dynamically. This is clearly an optimisation problem, that could eventually be solved with distributed heuristics expoiting a certain degree of locality. In order to do that, requirements and resource sets must be split into parts and mixed and matched in a distributed and incremental (partial) fashion. Requirements sould be expressed as predicates that evaluate to a certain degree of satisfaction [6]. Resources should be described without a predefined structure (descriptions could be of any type, not just memory, CPU and network). The languages needed to express requirements and resources, as well as good distributed resource matching algorithms are interesting research problems.

4.2 Data sharing in POP-C++ through ADHOC

POP-C++ implements asynchronous remote method invocations, using very basic system features, as TCP/IP sockets and POSIX threads. Instead of using those implemented parallel objects, POP-C++ could be adapted to use ADHOC objects. Calls to POP objects would be converted into calls to ADHOC objects. This would have the added advantage of being possible to somehow mix ADHOC applications and POP-C++ as they would share the same type of distributed object. This would as well add persistence to POP-C++ objects.

ADHOC objects are shared in a distributed system, as POP objects are. But they do not incorporate any concurrent semantics on the object side, neither their calls are asynchronous. In order to offer the same semantics, ADHOC objects (in the caller and in the callee side) would have to be wrapped in jackets, which would implement the concurrent semantics using something like POSIX threads. This does not appear to be a good solution.

4.3 Parallel POP-C++ components in the ASSIST framework

Currently, the ASSIST framework allows component programs to be developed with two type of components: **native** GRID.it components and **wrapped** legacy components. GRID.it components can either be sequential or parallel. They provide both a functional interface, exposing the computing capabilities of the component, and a non functional interface, exposing methods that can be used to control the component (e.g. to monitor its behaviour). They provide as well a **performance contract** that the component itself takes ensures by exploiting its internal autonomic control features implemented in the non functional code. Wrapped legacy components, on the other hand, are either CCM components or plain Web Services that can be automatically wrapped by the ASSIST framework tools to look like a GRID.it native component.

The ASSIST framework can be extended in such a way that POP-C++ programs can also be wrapped to look like GRID.it components and therefore be used in plain GRID.it component programs. As the parallelism exploitation patterns allowed in native GRID.it components are restricted to the ones provided by the ASSIST coordination language, POP-C++ components introduce in the ASSIST framework the possibility of having completely general parallel components. Of course, the efficiency of the POP-C++ components is completely in charge of the POP-C++ compiler/runtime environment. Some interesting possibilities also come in this case from the exploitation of object oriented programming techniques to implement the non functional part of the GRID.it component. In other words, trying to exploit full POP-C++ features to implement a customizable autonomic application manager providing the same non functional interface provided by ASSIST/GRID.it components.

5 Conclusion

The questions discussed in this paper entail a CoreGRID fellowship. All the possibilities described in the previous sections are currently being considered. The main focus of interest is clearly the integration of GEA as the POP-C++ launcher and resource manager. This will impose modifications on POP-C++ runtime library and new funcionalities for GEA. Both systems are expected to improve thanks to this interaction, as POP-C++ will profit from better resource discovery and GEA will implement a less restricted model. A running prototype is expected for the end of the year.

Further research on the matching model will lead to new approaches on expressing and matching application requirements and resource capabilities. This model should allow a distributed implementation that dinamically adapt the requirements as well as the resource availability, being able to express both ASSIST and POP-C++ requirements, and probably others.

A subsequent step can be a higher level of integration, using POP-C++ programs as GRID.it wrapped legacy components. This could allow to exploit full object oriented parallel programming techniques in ASSIST programs on grids. The implications of POP-C++ parallel object oriented modules on the structured model of ASSIST are not fully identified, especially due to the dynamic aspects of the objects created. Supplementary study has to be done in order to devise its real advantages and consequences.

References

1. M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzoloand M. Torquati, M. Vanneschi, and C. Zoccolo. The Implementation of ASSIST, an Environment for Parallel and Distributed Programming. In *Proc. of EuroPar2003*, number 2790 in "Lecture Notes in Computer Science". Springer, 2003.
2. M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High-Performance Grid Programming in GRID.it. In *Component modes and systems for Grid applications*, CoreGRID. Springer, 2005.
3. M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.
4. M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par 2005: Parallel and Distributed Computing*, number 3149 in "Lecture Notes in Computer Science". Springer Verlag, 2004.
5. M. Aldinucci and M. Torquati. Accelerating apache farms through ad-HOC distributed scalable object repository. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *10th Intl Euro-Par 2004: Parallel and Distributed Computing*, volume 3149 of *"Lecture Notes in Computer Science"*, pages 596–605, Pisa, Italy, August 2004. "Springer".
6. S. Andreozzi, P. Ciancarini, D. Montesi, and R. Moretti. Towards a metamodeling based method for representing and selecting grid services. In Mario Jeckle, Ryszard Kowalczyk, and Peter Braun II, editors, *GSEM*, volume 3270 of *Lecture Notes in Computer Science*, pages 78–93. Springer, 2004.
7. F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE Intl Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.
8. H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The Intl Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
9. Platform Computing Corporation. *Running Jobs with Platform LSF*, 2003.
10. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *Intl Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, 1997.
11. Object Management Group. *CORBA Components*, 2002.
12. R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.
13. T.-A. Nguyen and P. Kuonen. ParoC++: A requirement-driven parallel object-oriented programming language. In *Eighth Intl Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'03), April 22-22, 2003, Nice, France*, pages 25–33. IEEE Computer Society, 2003.
14. R. Raman, M. Livny, and M.H. Solomon. Resource management through multilateral matchmaking. In *HPDC*, pages 290–291, 2000.
15. M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications . *Parallel Computing*, 12, December 2002.
16. G. von Laszewski, B. Alunkal, K. Amin, J. Gawor, M. Hategan, and S. Nijsure. The Java CoG Kit User Manual. MCS Technical Memorandum ANL/MCS-TM-259, Argonne National Laboratory, March 14 2003.
17. T. Ylonen. SSH - secure login connections over the internet. In *Proceedings of the 6th Security Symposium*, page 37, Berkeley, 1996. USENIX Association.

Lightweight Grid Platform: Design Methodology

Rosa M. Badia[5], Olav Beckmann[2], Marian Bubak[7], Denis Caromel[3], Vladimir Getov[4], Stavros Isaiadis[4], Vladimir Lazarov[1], Maciek Malawski[6], Sofia Panagiotidi[2], Jeyarajan Thiyagalingam[4]

[1] Institute for Parallel Processing, Acad. G. Bonchev Str., Bl. 25-A, Sofia, Zip 1113, Bulgaria.
[2] Department of Computing, Imperial College London, London SW7 2AZ, U.K.
[3] I3S - Univ. de Nice Sophia Antipolis - CNRS URA 1376, INRIA, 2004 Rt. des Lucioles, BP 93,F-06902 Sophia Antipolis, Cedex, France.
[4] Harrow School of Computer Sciences, University of Westminster, Watford Road, Northwick Park, Harrow HA1 3TP, U.K.
[5] Departament d'Arquitectura de Computadors (DAC), Universitat Politcnica de Catalunya, Campus Nord - Mdul D6, C/ Jordi Girona, 1-3, E-08034 Barcelona, Spain.
[6] Institute of Computer Science, AGH, Mickiewicza 30, 30-059 Kraków, Poland.
[7] Academic Computer Centre – CYFRONET, Nawojki 11,30-950 Kraków, Poland.

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

Keywords Generic Grid, Light-Weight platform, components technology

1 Introduction

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

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

In [20], Thiyagalingam *et.al.* set out the design principles for designing a light-weight Grid platform. In this paper, we extend their techniques to design a

lightweight, generic Grid platform, by carefully analysing requirements and enabling technologies along with special attention to component technology. The key to our design philosophy is dynamic, on-demand pluggable component technology through which we hope to add and remove features on demand.

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

2 Component Technologies

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

- Common Component Architecture (CCA) [3]
- Fractal Component Model [2]
- Enterprise Grid Alliance Reference Model [4]

In the Common Component Architecture (CCA) [3], components interact using ports, which are interfaces pointing to method invocations. Components in this model define "provider-ports" to provide interfaces and "uses-ports" to make use of non-local interfaces. The enclosing framework provides support services such as connection pooling, reference allocation etc. Dynamic construction and destruction of component instances is also supported along with local and non-local binding. An interface description language (known as Scientific Interface Description Language, SIDL [19]) may be used to specify the interfaces and associated constraints which are then later compiled using a dedicated compiler (SIDL Compiler) to generate source code in a chosen programming language. These features provide seamless runtime interoperability between components. However, CCA does not strictly specify component composition and control mechanisms.

The Fractal Component Model [2] proposes a typed component model in which a component is formed from a controller and a content, and Fractal components may be nested recursively inside the content part. The control part provides a mechanism to control the behaviour of the content either directly or by intercepting interactions between Fractal components. The recursive nesting, sharing and control features support multiple configurations. Components have well-defined access points known as *interfaces*, which could either be client- or server-interfaces. Components interact through interfaces using *operations*, which are either one-way or two-way operations. The operation type determines the flow of operation requests and the flow of results, i.e. one-way operations correspond to operation requests only, whereas two-way operations correspond to operations with results being returned. As the controller part may be used to manipulate the behaviour of the content part, the various composition operations can be formulated on-demand. This feature, combined with sophisticated

binding technology, may be used to re-configure components and compositions dynamically.

The Enterprise Grid Alliance provides a reference model [4] with the intention of adopting Grid computing related technologies within the context of enterprise or business. The model classifies the components into layers and aligns the model with industry-strength requirements. Components in the reference model include hardware resources. Components can be associated with component-specific attributes to specify dependencies, constraints, service-level agreements, service-level objectives and configuration information. One of the key features that the reference model suggests is the life-cycle management of components which could be governed by policies and other management aspects.

3 Requirements for a Generic Light-Weight Platform

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

- MOCCA/H20 [12]
- ProActive (and other realizations of Fractal) [5]
- CORBA [15]
- ICENI [11]
- Ibis [18]
- GRID Superscalar [1]
- Enterprise Grid Alliance Reference Model [4]

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

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

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

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

2. **Static and dynamic metadata**
   In complex distributed computing environments, metadata plays an important role. Especially in component-based environments, it is often imperative to be able to extract metadata information from components in order to ensure efficient composition of components, satisfy quality of service requirements and provide the building blocks for the dynamic properties of the platform (reconfigurability, adaptability). For optimal application composition it is necessary to hold information about each available component. Static metadata can provide information pertaining to implementation, version, compatibility issues, performance characteristics, restrictions, accounting details and alike.
   At the other end we have dynamic metadata information: information pertaining to dynamic properties of components and resources. Keeping track of dynamic properties of components and resources is vital for satisfying quality of service and other service-level agreements. Further, dynamic metadata can be used for efficient component optimisation, checkpointing, recovery from failures, logging, accounting and reporting. Dynamic metadata can go beyond isolated components, and cover the application composition as a whole in order to support cross-component optimisation, application steering, runtime dynamic workload balancing etc.
3. **Dynamic deployment of components**
   There are many reasons why the platform should deploy components dynamically, including reaction to changes and demands in the system. This is possible, only if the platform is capable of introducing, replacing and removing components dynamically with minimal disruption.
4. **Reconfiguration and adaptivity**
   The platform should realistically model and synthesise resources in order to install or un-install dynamically additional features and services. Further, appropriate reaction to environmental conditions with the right exploitation of modelling, synthesis and deployment of services is also a necessary feature of the platform to guarantee resilience to failures.
   This is essentially a form of reconfiguration or the ability of the platform to self-organise itself to agree with the QoS issues, service-level agreements and service-level objectives. In supporting reconfiguration and adaptivity, the platform may utilise rules, embedded-knowledge and knowledge gathered across runs.
5. **Support for both client/server and P2P resource sharing**
   In the traditional client/server model of resource sharing, a broker module (or a querying module) performs match-making between user requirements and resources. In contrast, in a decentralised system, providers and consumers interact with more freedom without the intervention of brokering modules, i.e. in a peer-to-peer fashion. While a regulated centralised access

mechanism guarantees enforcement of fair policy, a peer-to-peer mechanism reduces associated overheads and improves response time and performance. Further, in an ad-hoc and mobile environment, registration activities may introduce unnecessary delays. In contrast, a peer-to-peer scheme does not require any such mechanism at the central level. We hypothesise that the platform should support both of these schemes to enable context-based support for resource sharing.

6. **On-demand, provider-centric service provision**
   The platform should support on-demand creation of services when needed by clients. However, making service provision more provider-centric ensures that the platform is freed from complex resource modelling and binding issues. In a provider-centric model, the provider enables service provision. This essentially frees the platform from performing unnecessary negotiations and coordination tasks.
7. **Minimal but sufficient security model**
   Maximised security, performance and simplicity of the platform are contradictory goals in design. Though the minimal security model may offer acceptable performance and may result in a light-weight platform, it can in practice be challenging to quantify the right level of "minimal security". Security requirements are often context-based. For example, in a trusted or isolated network, security measures can be bypassed in favour of performance and simplicity. In a collaborative network, such as the Internet, it is inevitable that security measures are tightened with minimal concern over performance issues. We intend to include support for single sign-on and delegation of credentials and mechanisms required in resource sharing environments which may span multiple administrative domains and pluggable support for any additional security features. This approach guarantees that the security model may evolve with context.
8. **Binding and coordination**
   Resources should be configurable by pluggability and reconfigurability of the platform, thus making possible the scenarios available in H2O. The roles of resource provider, component deployer and client should be separated, but they may possibly overlap. The platform should not mandate a specific mechanism for coordination and matching of users and providers. This should be left for pluggable discovery and brokering components. The presence of a centralised coordination point enables effective binding of resources, providers and users. However, such a centralised point can be a bottleneck in a loosely federated environment with no control. This would urge us to consider technologies for binding of resources, providers and users through a decentralised scheme with less negotiation for provision, utilisation and coordination of entities.
9. **Additional services**
   The platform should be able to incorporate additional services as requested by the environment. For example, a network environment may opt to bill the users during peak time (utility accounting), provide additional smarter discovery protocols at a small charge, automated backup services etc.

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

4 Evaluation of Existing Technologies

- **MOCCA/H2O**: MOCCA [12] is a lightweight distributed component platform, an implementation of the CCA framework built on top of the Java-based H2O resource sharing platform. MOCCA uses H2O [9] as a mechanism for creation of components on remote sites and uses RMIX [10] for communication between components. MOCCA takes advantage of the separation of the roles of resource provider and service deployer in H2O. Components in MOCCA can be dynamically created on remote machines. H2O kernels, where components are deployed, use the Java security sandbox model, giving a secure environment for running components. The extensible RMIX communication library allows using various protocols for communication, such as JRMP or SOAP, and also pluggable transport layers, including TCP, SSL, and JXTA [8] sockets for P2P environment.
- **ICENI and ICENI II**: ICENI [13, 6] is a Grid middleware infrastructure which includes methods for both resource management and efficiently deploying applications on Grid resources. The design philosophy of ICENI is based on high-level, component-based software construction, combined with declarative metadata that is used by the middleware to achieve effective execution. ICENI II [14] is a natural semantic evolution of ICENI, maintaining the architectural design of the original ICENI, but overcoming weaknesses in the current implementation, such as the implementation of ICENI on top of Web Services, decomposition of ICENI architecture into a number of separated composable toolkits and reduction of the footprint of ICENI on resources within the Grid.
- **ALiCE**: ALiCE is a lightweight Grid middleware which facilitates aggregation and virtualization of resources within an intranet and leveraging sparse resources through the Internet. The modularised, object-oriented nature of its implementation supports possible extensions and varying the levels of QoS, monitoring and security. The ALiCE architecture consists of multiple layers with the lowest, core layer providing resource discovery and system management using Java technologies. The second level layer, relying on the lowest layer, supports application development and deployment. The ALiCE runtime system consists of consumer, resource broker and a producer and task-farm manager which deploys and executes applications.
- **IBIS**: Ibis [18] is a Java-based Grid programming environment, aiming to provide portability, efficiency and flexibility. Ibis offers such programming models as traditional RMI (Remote Method Invocation), GMI for group communication, RepMI for replicated objects and Satin for solving problems using divide-and-conquer method. These components of Ibis are placed

on top of the Ibis Portability Layer (IPL), which allows various implementations of underlying modules, such as communication and serialisation, monitoring, topology discovery, resource management, and information services. IPL allows runtime negotiation of optimal protocols, serialisation methods, and underlying grid services, depending on the hardware and software configuration and requirements from higher layers.
Ibis focuses on various performance optimisations, to overcome the known drawbacks of Java. The optimisations include the serialisation of objects in RMI, avoiding of unnecessary copying of data during communication, and possibility of using native communication libraries e.g. for Myrinet.

- **CORBA**: Common Object Request Broker Architecture (CORBA [15]) is a middleware specification for large-scale distributed applications. An application in the CORBA architecture is composed of objects and the description of operations and functionalities of each and every object is utilised for providing support at the architecture level. Interface descriptions are used for communicating objects, transporting data and marshaling/un-marshaling methods calls. The IDL (Interface Description Language) definitions are language-independent, through mappings from a chosen programming language. The interfaces are compiled and mapped to the underlying programming language with a compiler provided by the ORB (Object Request Broker) system, which is the key to CORBA's interoperability. Method invocations on objects are handled transparently by the ORB, providing maximum abstraction. To capture dynamically and provide information regarding new objects, the ORB model provides a Dynamic Invocation Interface (DII) , which unifies the operations to all instances of an object. With DII, clients can construct the invocations dynamically by retrieving the object IDL interface from the registry.
- **GRID Superscalar**: GRID Superscalar is an Grid-unaware application framework focused on scientific applications. The definition of Grid-unaware applications in the framework of GRID Superscalar are those applications where the Grid (resources, middleware) is transparent at the user level, although the application will be run on a computational Grid. The key for GRID Superscalar applications is the identification of coarse grain functions or subroutines (in terms of CPU consumption) in the application. Once these functions or subroutines (called tasks in the GRID Superscalar framework) are identified, the GRID Superscalar system is able to detect at runtime data dependencies and the inherent concurrency between different instances of the tasks. Therefore, a data-dependence task graph is dynamically built, and tasks are executed on different resources on the Grid. Whenever possible (because data-dependencies and available resources allow) different tasks are executed concurrently, increasing application performance.
  The input codes for GRID Superscalar are sequential applications written in an imperative language, where a small number of GRID Superscalar API calls has been added. Another input that the user should provide is the IDL file, where the coarse grain functions/subroutines are identified by the user specifying their interface. A code generation tool uses the IDL file to

generate all the remaining files so that the application can be run on a Grid environment. This is combined with the deployment centre, which is graphical interface that enables to check the grid configuration and to automatically deploy the application in the grid. Optionally the user can also specify determined requirements of the tasks (resource, software, hardware) in a constraint specification interface. These requirements are matched at runtime by the GRID Superscalar library and the best resource that meets the requirements is selected to execute each task.

5 Use-Case Scenarios

In the final paper, we intend to include at least the following two use-case scenarios to illustrate better and capture the practical requirements from a user's point of view.

- GENIE: Grid ENabled Integrated Earth System Model
- Visualisation of Large Scientific Datasets

The GENIE (Grid ENabled Integrated Earth System Model) project [7] is an application which demonstrates the need for a scalable modular architecture. The project models the behaviour of large-scale thermohaline circulation, utilising various scientific modules corresponding to different environmental fragments. The case study would focus on componentizing the currently available serial solution for execution in a Grid platform. This task opens up a series of challenges including efficient componentization and composition, interface constraints, model-specific and resource-constrained simulations, real-time scheduling of operations, distributed execution and collection of large volumes of data.

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

The second use-case, Visualisation of Large Scientific Datasets, captures the requirements for the platform to manipulate interactively and visualise large volumes of data in a Grid environment. The large datasets are partitioned offline but the operations for visualisation are determined at runtime using a front-end, such as the MayaVi tool [17]. Visualisation of a given dataset typically involves processing a "visualisation pipeline" of domain-specific data analysis and rendering components. This happens before rendering and includes operations such as feature extraction or data filtering computations. Osmond *et al.* [16] describe the implementation of a "domain-specific interpreter" that allows visualisation pipelines specified from MayaVi to be executed in a distributed-memory parallel environment.

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

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

6 Conclusions

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

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

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

References

1. Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, Joseacute, M. Cela, and Rogeli Grima. Programming Grid applications with GRID superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.
2. E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the Seventh International Workshop on Component-Oriented Programming (WCOP2002)*, 2002.
3. CCA Forum Home Page. The Common Component Architecture Forum, 2004. `http://www.cca-forum.org`.
4. Enterprise Grid Alliance. Reference model. Technical Report Version 1.0, Enterprise Grid Alliance, 2005.
5. Matthieu Morel Francoise Baude, Denis Caromel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Italy*, volume 2888 of *LNCS*, pages 1226 – 1242. Springer, 2003.

6. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Journal of Parallel Computing*, 28(12):1753–1772, 2002.
7. GENIE. The Grid ENabled Integrated Earth system model project. `http://www.genie.ac.uk`, 2005.
8. Pawel Jurczyk, Maciej Golenia, Maciej Malawski, Dawid Kurzyniec, Marian Bubak, and Vaidy S. Sunderam. A system for distributed computing based on H2O and JXTA. In *Proceedings of the Cracow Grid Workshop, CGW'04, December 13–15, 2004*, pages 257–268, Kraków, Poland, 2005.
9. 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.
10. Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, and Aleksander Slomiński. RMIX: A multiprotocol RMI framework for java. In *Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 140–146, Nice, France, April 2003. IEEE Computer Society.
11. William Lee, Anthony Mayer, and Steven Newhouse. ICENI: An Open Grid Service Architecture implemented with Jini. In *SC2002: From Terabytes to Insight. Proceedings of the IEEE ACM SC 2002 Conference*. IEEE Computer Society Press, 2002.
12. Maciej Malawski, Dawid Kurzyniec, and Vaidy Sunderam. MOCCA – towards a distributed CCA framework for metacomputing. In *Proceedings of the 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2005)*, 2005.
13. Anthony Mayer, Andrew Stephen McGough, Nathalie Furmento amd Jeremy Cohen, Murtaza Gulamalim, Laurie Young, Ali Afzal, Steven Newhouse, and John Darlington. *Component Models and Systems for Grid Applications*, chapter ICENI: An Intergrated Grid Middleware to Support e-Science, pages 109–124. Springer Verlag, 2004.
14. Andrew Stephen McGough, William Lee, and John Darlington. ICENI II Architecture. In *UK e-Science All-Hands Meeting*, September 2005.
15. Object Management Group, Inc. CORBA, 2005. http://www.corba.org/.
16. K. Osmond, O Beckmann, A.J. Field, and P.H.J. Kelly. A domain-specific interpreter for parallelizing a large mixed-language visualisation application. To Appear in Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing.
17. Prabhu Ramachandran. MayaVi: A free tool for CFD data visualization. In *4th Annual CFD Symposium, Aeronautical Society of India*, August 2001. `mayavi.sourceforge.net`.
18. Rob van Nieuwpoort and Jason Maassen and Gosia Wrzesinska and Rutger F. H. Hofman and Ceriel J. H. Jacobs and Thilo Kielmann and Henri E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency - Practice and Experience*, 17(7-8):1079–1107, 2005.
19. S.Kohn, G. Kumfert, J. Painter, and C.Ribbens. Divorcing Language Dependencies from a Scientific Software Library. In *Proc. of the 10th SIAM Conf. on Parallel Processing for Sci. Comp.*, Portsmouth, USA, March 2001. SIAM.
20. Jeyarajan Thiyagalingam, Stavros Isaiadis, and Vladimir Getov. Towards Building a Generic Grid Services Platform: A Component Oriented Approach. In Vladimir Getov and Thilo Kielmann, editors, *Component Models and Systems for Grid Applications*, pages 39–46. Springer, 2005.

Classifier-Based Capacity Prediction for Desktop Grids

Artur Andrzejak[1], Patricio Domingues[2], Luis Silva[3,*]

[1]Zuse-Institute Berlin
Takustr. 7, 14195 Berlin, Germany
andrzejak@zib.de
[2]ESTG-Polytechnic Institute of Leiria
2411-901 Leiria, Portugal
patricio@estg.ipleiria.pt
[3]CISUC - Centre for Informatics and Systems of the University of Coimbra
3030-290 Coimbra, Portugal
luis@dei.uc.pt

Abstract. Availability of resources in desktop grids is characterized by high dynamicity, a consequence of the local (owner) control policies in such systems. The efficient usage of desktop computers can therefore greatly benefit from prediction methodologies, as those help to estimate the short-term behavior of resources and to take the appropriate preventive actions. We present a prediction study performed on institutional desktop pool and discuss a framework for automated behavior prediction. Several classifier-based forecast algorithms are evaluated over two traces collected from 32 machines of two classrooms of an academic environment. Results show that prediction approaches produce meaningful results in the context of desktop grids and can be used for more efficient and dependable computing in these environments. Moreover, the applied prediction methodology - classification algorithms - allow for computationally inexpensive real time predictions.

1 Introduction

Grids comprised of networked desktop computers, commonly referred as desktop grids, are becoming increasingly attractive for performing computations. It is well-know that desktop machines used for regular activities, ranging from electronic office actions (text processing, spreadsheets and other document preparation) to communication (e-mail, instant messaging) and information browsing have very low resource usage. Indeed, most computing activities depending on human interactive input barely load the machines yielding high percentage of resource idleness, namely CPU, making these resources attractive for harvesting [1].

Popularity of harvesting desktop resources can be attested by the numerous and popular public computing projects [2], like the well-known SETI@home [3]. The success of desktop grid computing has fostered the development of middleware yield-

* Authors in alphabetical order

ing several frameworks, ranging from academic projects like Condor [4], BOINC [5] and XtremWeb [6], to commercial solutions like United Devices [7], to name just a few. The availability of these platforms has promoted the setup and use of desktop grids, which in turn has fueled further development and refinement of desktop grids middleware.

Desktop grids are not restricted to public computing projects. In fact, many institutions like academics or corporate own respectable amount of desktop computers (hundredths or even thousands), therefore holding an impressive computing power if properly harnessed. The availability of desktop grid middleware has made possible the setup and exploitation of institutional desktop grids with reduced effort. These institutional desktop grids can provide demanding users (researchers, staff, etc.) inexpensive computing power. In this study, we focus on institutional desktop grids comprised of resources connected by a local area network, like, for instance, all desktop machines of an academic campus or existing at a company's facilities.

The major drawback of desktop grids lies in the volatility of resources. Indeed, since desktop machines are primarily devoted to their owners, resources might suddenly alter their availability statuses. Thus, efficiency of use of desktop grid environments can greatly benefit with prediction of resources availability. In fact, if a given machine or scheduler knows, with high probability, of a failure in a short timescale future, actions can be taken to eliminate or at least minimize the effects of the predicted failure. For instance, the application can be migrated or replicated to another machine whose prediction indicates higher probability to survive and/or higher amount of resources available for harvesting. Therefore, accurate resource prediction can substantially increase usage efficiency of desktop grid based systems.

The main motivation for this work was to assess the feasibility and accuracy of predicting desktop resources availability from academic classrooms based on the resource usage traces collected from desktop machines. The results show that indeed even such a highly volatile environment allows for meaningful and robust prediction results. Moreover, the deployed classification algorithms known from data mining have low computational cost and allow for online usage without significant resource overhead. These results show that prediction methods can provide effective support for better usage of resources in desktop Grids and for improving their dependability.

The remainder of this paper is organized as follows. Section 2 describes related work, while section 3 presents the trace collecting utility and the prediction framework. Section 4 details the prediction experiment, with results being discussed in section 5. Finally, section 6 concludes the paper.

2 Related work

Work on resource demand prediction includes calendar methods [8]. In these approaches, a fixed period (e.g. 1 day) is assumed, and a probabilistic profile for each chunk of the period is computed. One drawback of these methods is that the period is determined in an arbitrary way. Here the Fast Fourier Transformation can be used to determine the essential periodicities. More advanced approaches to demand prediction stem from econometrics and use time series models based on auto-regression and

moving averages such as ARIMA and ARFIMA [9]. Another class of prediction methods (used in this study) deploys classification algorithms known from data mining [10]. Here a set of examples which include some function values (attributes) of past samples together with the correct classification (i.e. prediction value) is first presented to the classifier and used to build an internal model. Subsequent requests to the classifier with the values of the analogous attributes yield a prediction as the response.

The most advanced framework for resource prediction in computer systems is the Resource Prediction System (RPS) [11]. It is a toolkit for designing, building and evaluating systems that predict dynamic behavior of resources in distributed systems. For analyzing and predicting data series, RPS offers a library and several tools supporting several models like MEAN, NEWTON, ARIMA and wavelet methods. The main disadvantages of RPS are its focus on very short-term predictions, lack of correlation analysis, and high computational demand for ARIMA model creation.

The Network Weather Service (NWS) [12] is a distributed framework that aims to provide short-term forecasts of dynamically changing performance characteristics of a distributed set of computing resources. NWS operates through a set of performance sensors ("monitors") from which it gathers readings of instantaneous conditions. Performance forecasts are drawn from mathematical analysis of collected metrics. A limitation of NWS lies in its support restricted to Unix environments, especially since a high percentage of desktop machines run Windows based operative systems.

3 The collection and prediction frameworks

In this section, we briefly introduce the two frameworks employed in our study: Distributed Data Collector (DDC) and OpenSeries. The former is used to collect the resource usage traces while the latter is the framework that offers the data mining capabilities for time series analysis of the resource traces. Thus, roughly put, DDC collects resource usage traces that are then fed into the OpenSeries module for generating resource usage prediction.

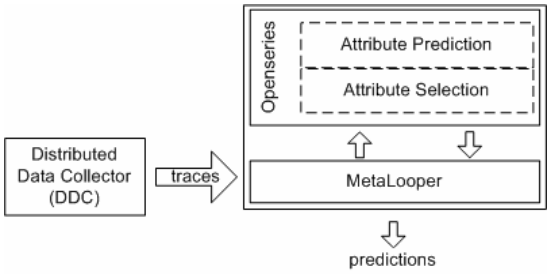


Figure 1: Software frameworks and the flow of data

It is important to note that DDC and OpenSeries are decoupled systems that can be run independently of each other. In fact, DDC can be used for wider applications than resource usage trace collection, and input traces for OpenSeries only needs to respect the Comma Separated Value (CSV) format or similar, permitting that OpenSeries

receives input from other traces collector software. Figure 1 depicts the relationship between the trace collector and the data analyzer and the predictor modules.

3.1 Distributed Data Collector

DDC is a framework to automate repetitive executions of console applications (probes) over a set of networked Windows personal computers [13]. DDC schedules the periodic execution of software probes in a given set of machines. The execution of probes is carried out remotely, that is, the probe binary is executed at the remote machine, requiring only the appropriate access credentials. All executions of probes are scheduled by DDC's central coordinator host, which is a normal PC. A probe is a mere win32 console application that uses its output channels (both *stdout* and *stderr*) to communicate its results. One of DDC tasks is precisely to capture the output of the probe and to store it at the coordinator machine. Note, that from the perspective of the probe, the execution occurs locally, that is, the probe is unaware of being executed remotely. Additionally, DDC allows the probe's output to be processed by so-called *post-collecting code* which is supplied by DDC's user and specific to a probe. The purpose of the post-collecting code is to permit analysis of probe's output immediately after its execution, so that relevant data can be extracted.

A key feature of DDC is that no software needs to be installed at remote machines. Indeed, the framework uses Windows capabilities for remote execution, requiring solely the credentials to access remote machines.

For the purpose of collecting usage traces of desktop resources suitable for time series analysis, DDC was fitted with an appropriate probe that collects, for every execution at a remote machine, the current timestamp, CPU idleness percentage, memory load (percentage of physical memory used), and machine uptime, amongst other metrics. Collected data are appended to the trace file corresponding to the remote machine. At the end of a timestamp, after all remote machines have been probed, an entry is appended to the so-called trace index file, stating which machines responded positively to the probe and which ones were not probed, because of being not accessible (powered off or disconnected from the network). The trace index file aggregates, for every timestamp, the list of available machines and can be used for replaying the trace.

3.2 The OpenSeries framework

OpenSeries [14] is an extensible framework for time series analysis, modeling, and prediction. It is a collection of packages which provide data structures, persistency layer, and a deployment engine. Currently only off-line analysis and prediction is implemented, but the online (real time) modules are in work. The framework interfaces with the Weka 3.4 data mining library, offering the user access to a variety of classification algorithms, data filters, attribute selection techniques and clustering algorithms. Additionally, OpenSeries currently provides multivariate ARIMA-type

algorithms [9] known from econometrics, neuro-fuzzy classifiers, and genetic algorithms which can be used for model fitting or other parameter tuning.

OpenSeries can be controlled and configured by MetaLooper [15], a component-based batch experiment configuration and execution framework. It facilitates the configuration of batch experiments, while promoting code reuse. MetaLooper permits to configure user's Java-code organized in a component-oriented fashion into trees that are orderly executed by the framework. Experiments and associated parameters are defined through XML, freeing MetaLooper's users from the burden of having to implement configuration related elements in Java.

4 Evaluation

Our prediction methodologies were evaluated using two groups of metric traces (henceforth, trace A and trace B). Each trace represents the resource usage of two classrooms with 16 machines each, thus totaling 32 machines. Although the classrooms are primarily devoted to classes, during off-classes any student from the academic institution can access the machines for performing practical assignments (coding, writing report and so on), communicate through e-mail and browse the web.

Both traces were collected with a two-minute interval between consecutive samples in a machine. It is important to note that no policy related to the powering off of machines exists. In fact, users are advised but not ordered to power off machines when they leave the machines. On weekdays, classrooms remain open 20 hours per day, closing from 4 am to 8 am. On Saturdays, classrooms open at 8 am and close at 9 pm until the next Monday.

Trace A represents 39 consecutive days (25/11/2004 to 3/01/2005) and contains 27000 samples, while trace B corresponds to 17 consecutive days (31/01/2005 to 16/02/2005) and has 12700 samples. Contrary to trace A that was collected on a period with classes, trace B corresponds to an exam-period, without classes.

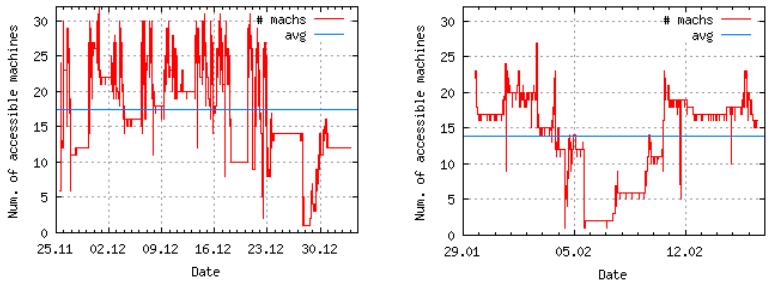


Figure 2: Count of machines over time for trace A (left) and trace B (right). The horizontal line represents the average count of accessible machines (17.42 for trace A, 13.94 for trace B)

Figure 2 shows the count of alive machines over time for trace A (left) and trace B (right). The weekday/weekend periods are distinguishable, with weekends exhibiting a stable number of accessible machines, identifiable by the flat sections of the plots. Also visible in the plot is that trace A presents a higher number of accessible machines than trace B. In fact, accordingly to the average count of accessible machines, trace A has 3.48 more powered on machines than trace B (17.42 versus 13.94).

4.1 Prediction methodologies

In our study we deployed the classification algorithms known from data mining instead of the typically used ARIMA-based methods. This approach turned out to be successful in this setting, and features the following advantages:

- discretized numerical prediction (such as expected CPU load) can be combined with binary-type predictions (such as intrusion alert).
- the inter-trace correlations can be easily exploited without excessive increase of computational cost contrary to the case of multivariate ARIMA predictions.
- the computational cost for training of the predictor is considerably lower than for the ARIMA method (even without using inter-trace correlations).
- a variety of algorithms can be used, including “lazy” methods which incur little computational effort during the training phase yet more during the prediction, or “non-lazy” which are more computationally demanding during training yet fast during prediction (useful for repetitive predictions based on the same model).

4.1.1 Prediction through classification

To predict continuous signal using classifiers, signal value for a given sample has to be discretized into a fixed number of levels. For example, if the CPU usage signal ranges from 0% and 100%, and the sufficient prediction accuracy is on the order of 20%, than five levels with intervals [0%-20%[, .., [80%-100%] should be used. A classifier returns the index of one these levels as the prediction value.

In general, a classifier requires a set of training examples to build a prediction model. Each such training example consists of a tuple of attribute values and the correct class (signal level in our setting) to be predicted. Each attribute is either a raw input data (past sample value), or some function thereof, such as a moving average or first derivative. For exploiting inter-trace correlations, the attributes may be also based on traces which are different from the target trace. For example, when predicting the availability of machine X (target), we could use the moving averages of the CPU load of other machines as the attributes.

4.1.2 Attribute selection

It is known that a too large number of attributes as classifier input decreases the prediction accuracy [10] due to the over fitting phenomena. Especially when using inter-trace correlations and functions of raw data, the number of potential attributes could very large (on the order of few thousands in the case of our study). To reduce the number of the attributes, attribute selection methods are used.

However, in the case of using inter-trace correlations, the initial number of attributes to select from is so large that it incurs too heavy computational costs (some attribute selection algorithms are quadratic in the number of initial attributes). Therefore, we have developed a two-stage process which first selects the relevant (correlated) machines (phase A), then computes a pool of functions on traces of these ma-

chines (phase f), and finally selects the final attributes from the pool of functions (phase B). Since phase A is essentially a trace correlation analysis with low running time, we can specify a large set of machines (on the order of 100) as a potential input.

4.1.3 Walk-forward evaluation

The prediction testing is performed in a walk-forward mode. A fitting interval of size F is used for training the models (building the classifier models). An adjacent test interval of size T is used for the out-of-sample evaluation of the model. Subsequently, both intervals are moved forward by F, and the process is repeated. This procedure allows for change-adaptive model selection yet avoids any over fitting problems.

4.2 Experiments

In our study we used three prediction targets:

- percentage of machine idleness (idle percentage)
- percentage of free virtual memory (memory load)
- machine availability, i.e. whether it was switched on or off (availability).

The first target had original value range [0,..,100] and has been discretized into five levels. The second target had also values between 0 and 100, and four discretization levels have been used. Machine availability was either 0 or 1, and consequently has been assigned two levels.

The attributes used in the attribute selection and prediction included the raw data and simple moving averages (SMA) thereof, with length from 5 to 60 samples. Also calendar functions of the timestamps such as hour of the day, day of the week etc. were included. As the lead time (the offset of the future time instant for which prediction is made) we have chosen 30 minutes as a reasonable time to perform management actions in a desktop grid. We also tested other lead times (from 15 to 120 minutes) which yielded similar accuracies.

At the end of the preliminary attribute selection phase A (resource selection) we have taken the 5 most relevant “sources” (pairs machine/metric) and used the functions thereof as input for the final selection phase. At the end of this second phase, 10 most relevant attributes have been used for the prediction.

All targets have been predicted using the following five classification algorithms, with algorithm parameters having the default values in the Weka 3.4 library:

- Naive Bayes (Bayes)
- complement class Naive Bayes classifier (CompBayes)
- John Platt's sequential minimal optimization algorithm for training a support vector classifier (SMO)
- k-nearest neighbors classifier (IBk)
- C4.5 decision tree classifier (treesJ48).

The evaluation of the prediction was performed in the walk-forward mode, with 1 week as the fitting time interval F, and 3 days as the test time interval T. Within each test interval, the predictions have been performed for each of the 2160 samples.

5 Results

In the following we state the results of our study and interpret them. All running times have been measured on a 3 GHz Pentium 4 machine with hyperthreading and 1.5 GByte RAM. We used the Java 1.5 client VM under Suse Linux.

5.1 Attribute selection results

The attribute selection phase has been performed for each of the 32 machines and each of the 3 targets, in total 96 times (for each of the traces A and B). The resulting attributes showed very little inter-machine correlations, probably due to the fact that there is no strict power on/off policy. That is, users are free to choose the machine they want to work at and are not obliged to power off the machine they worked at. Moreover, when a class starts, some of machines will be already powered on, while others need to be switched on.

The computing time for the attribute selection is non-negligible, however the relevant attributes need to be computed only once and recomputed only if the usage patterns significantly change. For example, for the (longer) trace A the total selection time for all 96 targets was below 4 hours, or 2.5 minutes per target.

5.2 Prediction results

As noted above, five different classification algorithms have been used. Overall, the Support Vector Machines classifier (SMO) has performed best. In some cases the simple Naive Bayes algorithm could achieve only slightly lower accuracy, but required significantly less computation time. We have used for evaluation the Mean Squared Error (MSE) of the prediction, comparing it to the variance of the original signal. In Figure 3 we compare the classifier accuracy in terms of the sum of MSEs over all machines in a trace.

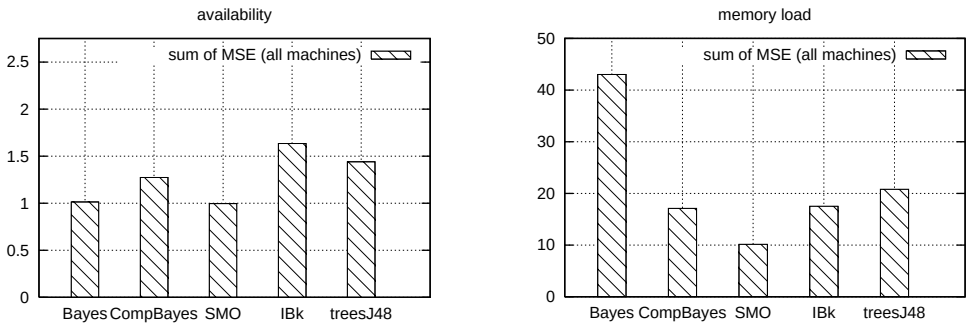


Figure 3: Comparison of the classifier accuracy: trace A (left), trace B (right)

The results of predictions using SMO for the memory load are shown in Figure 4, those for percentage of CPU idleness in Figure 5, and those for the machine availability in Figure 6. In the whole study, there is only one (non-zero) case where the MSE is higher than the signal variance. This indicates high robustness of the prediction and reliability of the method.

In general, the prediction results for trace A are better. As visible in Figure 2, some machines in trace B have not been used frequently during the data collection period. This yields low signal variances for those machines, which makes the MSE to appear large in comparison.

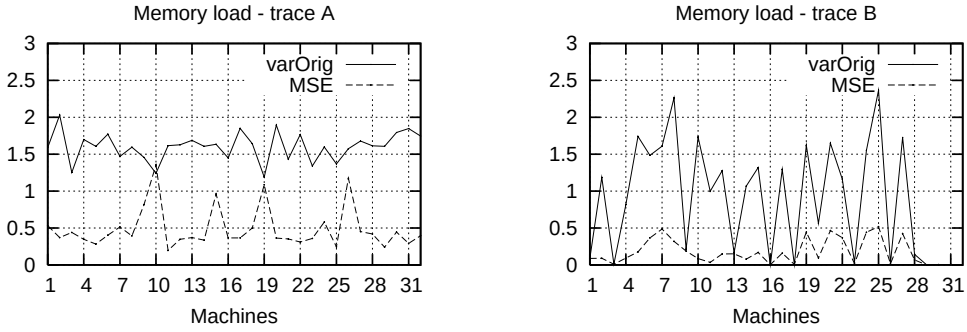


Figure 4: Prediction results for memory load for traces A and B

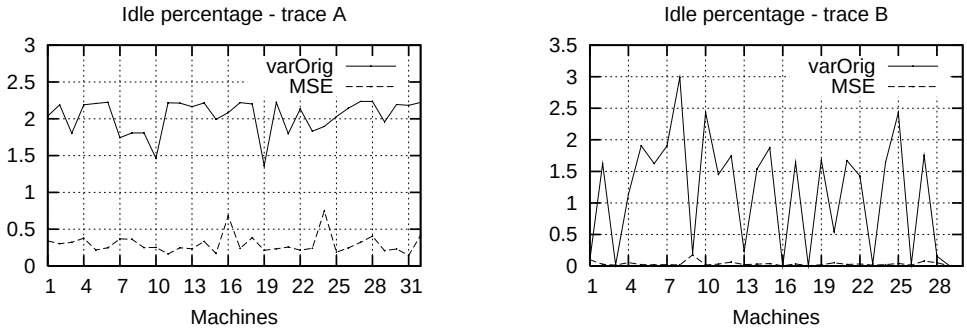


Figure 5: Prediction results for CPU idleness percentage for traces A and B

The cumulative running time for trace A was below 9 hours for all 32x3x5 = 480 prediction cases. Each case included 11 times the creation of the classifier model (there are (39-7)/3= 11 phases in walk-forward test) and approximately 22000 predictions. On average, a single case required 67.5 seconds, or 3 msec per prediction (amortized). This is the average time over all 5 classifiers; specific classifiers can incur larger computational cost (SMO is likely to need more time than Naive Bayes). However, since a new prediction would be required at most every 2 minutes in the real-time case, our approach overall incurs negligible computational overhead.

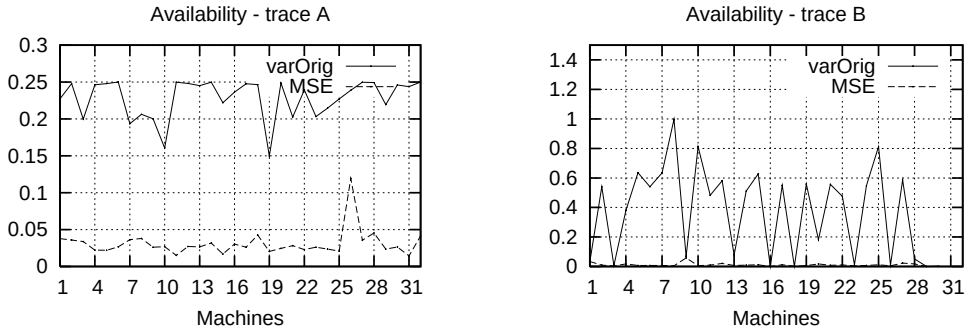


Figure 6: Prediction results for machine availability for traces A and B

6 Conclusions

This study has shown that even highly dynamic environments such as desktop pools allow for meaningful predictions of a variety of metrics. Primary applications of this technique are in the domain of desktop grid schedulers and resource managers. Those can use predictions for job placements which require least migration, provide highest failure or interruption resilience, or are least intrusive for resource owners.

The prediction results achieved in the study are robust and significant. There are consistent differences in the accuracy of different classification algorithms, with SMO (Support Vector Machines) performing best. Inexistence of correlation is possibly due to the lack of a strict power off/on policy.

Acknowledgements

This research work is carried out in part under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265). Artur Andrzejak would like to thank Mehmet Ceyran and Ulf Hermann for part of the implementation work.

References

[1] P. Domingues, P. Marques, and L. Silva, "Resource Usage of Windows Computer Laboratories," ICPP Workshops 2005, Oslo, Norway, 2005.

[2] J. Bohannon, "Grassroots supercomputing," *Science*, pp. 810-813, 2005.

[3] Seti@Home, "Seti@Home (http://setiathome.ssl.berkeley.edu/)," 2005.

[4] Condor, "Condor Project Homepage (http://www.cs.wisc.edu/condor/),".

[5] D. Anderson, "BOINC: A System for Public-Resource Computing and Storage,", 5th IEEE/ACM Workshop on Grid Computing, Pittsburgh, USA, 2004.

[6] G. Fedak, C. Germain, V. Neri, and F. Cappello, "XtremWeb: A Generic Global Computing System," 1st CCGRID'01, Brisbane, 2001.

[7] UnitedDevices, "United Devices, Inc. (http://www.ud.com)."

[8] J. Hollingsworth and S. Maneewongvatana, "Imprecise Calendars: An Approach to Scheduling Computational Grids.," Int. Conf. on Distributed Comp. Systems, 1999.

[9] W. Enders, *Applied Econometric Time Series*, 2nd ed: Wiley Canada, 2003.

[10] I. H. Witten and F. Eibe, *Data mining: practical machine learning tools and techniques with Java implementations*: Morgan Kaufmann Publishers, 2000.

[11] P. Dinda and D. O'Hallaron, "An extensible toolkit for resource prediction in distributed systems," Carnegie Mellon University CMU-CS-99-138, 1999.

[12] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Future Generation Computing Systems*, vol. 15, pp. 757-768, 1999.

[13] P. Domingues, P. Marques, and L. Silva, "Distributed Data Collection through Remote Probing in Windows Environments," 13th Euromicro Parallel, Distributed and Network-Based Processing, Lugano, Switzerland, 2005.

[14] A. Andrzejak, M. Ceyran, and U. Hermann, "OpenSeries - a Time Series Analysis Library," (unpublished) 2005.

[15] A. Andrzejak and M. Ceyran, "MetaLooper - a framework for configuration and execution of batch experiments with Java (http://www.zib.de/andrzejak/ml/text.en.html)," 2005.

A Feedback-based Approach to Reduce Duplicate Messages in Unstructured Peer-to-Peer Networks

Charis Papadakis[1], Paraskevi Fragopoulou[1], Elias Athanasopoulos[1], Marios Dikaiakos[2], Alexandros Labrinidis[3] and Evangelos Markatos[1]

[1] Institute of Computer Science, Foundation for Research and Technology-Hellas
P.O. Box 1385, 71 110 Heraklion-Crete, Greece
{adanar, fragopou, elathan, markatos}@ics.forth.gr

[2] Department of Computer Science, University of Cyprus, P.O. Box 537, CY-1678 Nicosia, Cyprus
mdd@ucy.ac.cy

[3] Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260, USA
labrinid@cs.pitt.edu

Abstract. Unstructured P2P systems have used flooding as their prevailing resource location method. Each node forwards each incoming query message to all of its neighbours until the query propagates up to a predefined maximum number of hops away from its origin. Although this algorithm has excellent response time and is very simple to implement, it creates a large volume of unnecessary traffic in today's Internet because each node may receive the same query several times through different paths. We propose an innovative technique, the *feedback-based approach* that aims to improve the scalability of flooding. The main idea behind our algorithm is to monitor the ratio of duplicate messages transmitted over each network connection, and to avoid forwarding query messages over connections whose ratio exceeds some predefined threshold. Through extensive simulation we show that this algorithm exhibits significant reduction of traffic in random and small-world graphs, the two most common types of graph that have been studied in the context of P2P systems, while conserving network coverage.

1 Introduction

In unstructured P2P networks, such as Gnutella and KaZaA, each node is directly connected to a small set of other nodes, called neighbors. Most of today's commercial P2P systems are unstructured and rely on random overlay networks [7,9]. Unstructured P2P systems have used flooding as their prevailing resource location method [7,9]. A node looking for a file issues a query which is broadcasted in the network. An important parameter in the flooding algorithm is the Time-To-Live or TTL. The TTL indicates the number of hops away from its source a query should propagate. The node that initiates the flooding sets the query's TTL to a small positive integer, smaller than the diameter of the network. Each receiving node decreases by one the query TTL value before broadcasting it to its neighbors. The query propagation terminates when its TTL reaches zero.

The basic problem with the flooding mechanism is that it creates a large volume of unnecessary traffic in the network mainly because a node may receive the same queries multiple times through different paths. The reason behind the duplicate messages is the existence of cycles in the underlying network topology. Duplicates constitute a large percentage of the total number of messages generated during flooding. In a network of N nodes and average degree d and for TTL value equal to the diameter of the graph, there are N(d-2) duplicate messages for a single query while only N-1 messages are needed to reach all network nodes. The TTL was incorporated in the flooding algorithm in order to reduce the number of messages produced thus reducing the overall network traffic. Since the paths traversed by the flooding messages are short, there is a small probability that those paths will form cycles and thus generate duplicates. However, as we will see below, even this observation is not valid for small-world graphs. Furthermore, a small TTL value can reduce the *network coverage* defined as the percentage of network nodes that receive a query.

In an effort to alleviate the large volumes of unnecessary traffic produced during flooding several variations have been proposed in the literature [12]. Most of these rely on randomly or selectively propagating the query messages to a small number of each node's neighbours. The neighbour selection criteria is the number of responses received, the node capacity, or the link latency. Although these methods succeed in reducing excessive network traffic, they usually incur significant loss in network coverage, meaning that only a small part of the network's nodes are queried, thus a much smaller

number of query answers are returned to the requesting node. This can be a problem especially when the search targets rare items for which often no response is returned. Other search methods such as random walkers or multiple random walkers suffer from slow response time.

Aiming to alleviate the excessive network traffic problem while at the same time maintain high network coverage, in this paper, we devise an innovative technique, the feedback-based algorithm, that attacks the problem by monitoring the number of duplicates on each network connection and trying to forward queries over connections that do not produce an excessive number of duplicates. During an initial and relatively short warm-up phase, a feedback is returned for each duplicate that is encountered on an edge to the upstream node. Following the warm-up phase each node decides to forward incoming query messages on each of its incident edges based on whether the percentage of duplicates on that edge during the warm-up phase does not exceed some predefined threshold value. We show through extensive simulation, for different values of the parameters involved, that this algorithm is very efficient in terms of traffic reduction in random and small-world graphs, the two most common types of graph that have been studied in the context of P2P systems, while the algorithm exhibits minor loss in network coverage. Furthermore, a restricted version of the algorithm which gives the best results does not require any protocol modification.

The remainder of this paper is organized as follows: Following the related work section, the feedback-based algorithm is presented in Section 3. The two most common types of graphs that were studied in the context of P2P systems, and on which we conducted our experiments, are presented in Section 4. The simulation details and the experimental results on static graphs are presented in Section 5. Finally, the algorithm's behavior on dynamic graphs, assuming that nodes can leave the network and new nodes can enter at any time, is presented in Section 6. We conclude in Section 7 with a summary of the results.

2 Related Work

Many algorithms have been proposed in the literature to alleviate the excessive traffic problem and to deal with the traffic/coverage trade-off [12]. One of the first alternatives to be proposed was *random walk*. Each node forwards each query it receives to a single neighboring node chosen at random. In this case the TTL parameter designates the number of hops the walker should propagate. Random walks produce very little traffic, just one query message per visited node, but reduce considerably network coverage and have long response time. As an alternative *multiple random walks* have been proposed. The node that originates the query forwards it to k of it neighbors. Each node receiving an incoming query transmits it to a single randomly chosen neighbor. Although compared to the single random walk this method has better behavior, it still suffers from low network coverage and slow response time. Hybrid methods that combine flooding and random walks have been proposed in [5].

In another family of proposed algorithms query messages are forwarded not randomly but rather selectively to part of a node's neighbors based on some criteria or statistical information. For example, each node selects the first k neighbors that returned the most query responses, or the k highest capacity nodes, or the k connections with the smallest latency to forward new queries [6]. A somewhat different approach named *forwarding indices* [2] builds a structure that resembles a routing table at each node. This structure stores the number of responses returned through each neighbor on each one of a pre-selected list of topics. Other techniques include query caching, and the incorporation of semantic information in the network [3,10,14].

The specific problem we deal with in this paper, namely the problem of duplicate messages, has been identified and some results appear in the literature. In [13] a randomized and a selective approach is adopted and each query message is send to a portion of a node's neighbors. The algorithm is shown to reduce the number of duplicates and to maintain network coverage. The performance of the algorithm is demonstrated on graphs of limited size. In another effort to reduce the excessive traffic in flooding, Gkatsidis and Mihail [5] proposed to direct messages along edges which are parts of shortest paths. They rely on the use of PING and PONG messages to find the edges that lie on shortest paths. However, due to PONG caching is this not a reliable technique. Furthermore, their algorithm degenerates to simple flooding for random graphs, meaning that in this case no duplicate messages are eliminated. Finally, in [8] the authors propose to construct a shortest paths spanning tree rooted at each network node. However, this algorithm is not very scalable since the state each network node has to keep is in the order of O(Nd), where N is the number of network nodes and d its average degree.

3 The Feedback-based Algorithm

The basic idea of the feedback based algorithm is to identify edges on which an excessive number of duplicates are produced and to avoid forwarding query messages over these edges. In the algorithm's warm-up phase, during which flooding is used, a feedback message is returned to the upstream node for each duplicate message. The objective of the algorithm is to count the number of duplicates produced on each edge during this phase and subsequently, during the execution phase, to use this count to decide whether to forward a query message over an edge or not.

In a static graph, a query message transmitted over an edge is a duplicate if this edge is not on the shortest path from the origin to the downstream node. One of the key points in the feedback-based algorithm is the following: Each network node A forms groups of the other nodes, and a different count is kept on each one of A's incidents edges for duplicate messages originating at nodes of each different group. The objective is for each node A to group together the other nodes so that messages originating at nodes of the same group either produce many duplicates or few duplicates on each one of A's incident edges. An incident edge of some node A that produces only a few duplicates for messages originating at nodes of a group belongs to many shortest paths connecting nodes of this group to the downstream node. An incident edge of node A that produces many duplicates for messages originating at nodes of a group belongs to few shortest paths connecting nodes of this group to the downstream node. Notice that if all duplicate messages produced on an edge were counted together (independent of their origin), then the algorithm would be inconclusive. In this case the duplicate count on all edges would be the almost the same since each node would receive the same query though all of its incident edges. The criteria used by each node to group together the other nodes are critical for the algorithm's performance and the intuition for their choice is explained below.

A sketch of the feedback-based algorithm is the following:

- Each node A groups together the rest of the nodes according to some criteria.
- During the warm-up phase, each node A keeps a count of the number of duplicates on each of its incident edges, originating at nodes of each different group.
- Subsequently, during the execution phase, messages originating at nodes of a group are forwarded over an incident edge e of node A, if the percentage of duplicates for this group on edge e during the warm-up phase is below a predefined threshold value.

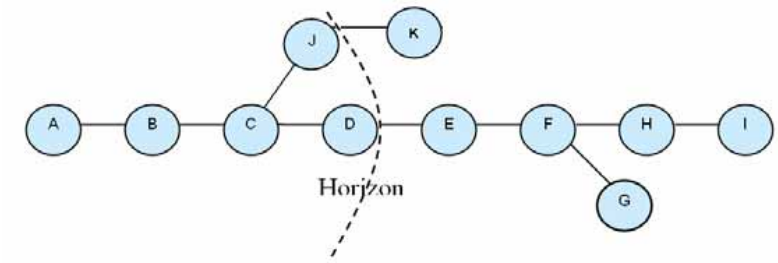


Fig. 1. Illustration of the horizon criterion for node *A* and for horizon value 3

Two different grouping criteria, namely, the hops, the horizon, and a combination of them horizon+hops are used that lead to three variations of the feedback-based algorithm.

- **Hops criterion:** Each node A keeps a different count on each of its incident edges for duplicates originating k hops away (k ranges from 1 up to the graph diameter). The intuition for this choice is that, as we will see below, in random graphs small hops produce few duplicates and large hops produce mostly duplicates. Thus, messages originating at close by nodes are most probably not duplicates while most messages originating at distant nodes are duplicates. In order for this grouping criterion to work each query message should store the number of hops traversed so far.
- **Horizon criterion:** The horizon is a small integer, smaller than the diameter of the graph. A node is in the horizon of some node A if its distance in hops from A is less than the horizon value, while all other nodes are outside A's horizon, Fig. 1. For each node inside A's horizon a different count is kept by A on each of its incident edges. Duplicate messages originating at nodes outside A's horizon are added up to the count of their entry node in A's horizon. For example, in Fig. 1, duplicates produced by queries originating at node K are added up to the counters kept for node J, while duplicates produced by queries originating at nodes E,F,G,H,I are added up to the counters kept for node D. The intuition for the choice of this criterion is that shortest paths differ in the first hops and

when they meet they follow a common route. For this criterion to be effective a message should store the identities of the last k nodes visited, where k is the horizon value.

- **Horizon+Hops criterion:** This criterion combines the two previous. Duplicates are counted separately on each one of A's incident edges for each node in A's horizon. Nodes outside A's horizon are grouped together according (1) to their distance in hops from A and (2) to the entry node of their messages in A's horizon.

Three variations of the feedback-based algorithm are presented based on the grouping criteria used. The algorithm using the hops criterion is show below:

Feedback-based algorithm using the Hops criterion

1. **Warm-up phase**
   - **a.** Each incoming non-duplicate query message is forwarded to all neighbors except the upstream one.
   - **b.** For each incoming duplicate query message received, a duplicate feedback is returned to the upstream node.
   - **c.** Each node A, for each incident edge e, counts the percentage of duplicate feedbacks produced on edge e for all queries messages originating k hops away. Let us denote this count by $C_{e,k}$
2. **Execution phase**
   - **a.** Each node A forwards an incoming non-duplicate query message that originates k hops away over its incident edges e if the count $C_{e,k}$ does not exceed a predefined threshold.

For the hops criterion to work each query message needs to store the number of hops traversed so far. The groups formed by node A in the graph of Fig. 1 according to the hops criterion are shown in Table 1.

The algorithm using the horizon criterion is shown below:

Feedback-based algorithm using the Horizon criterion

1. **Warm-up phase**
   - **a. & b.** Same as in Hops criterion
   - **c.** Each node A, for each incident edge e, counts the percentage of duplicates produced on edge e for all query messages originating at a node B inside the horizon, or entered the horizon at node B. Let us denote this count by $C_{e,B}$.
2. **Execution phase**
   - **a.** Each node A forwards an incoming non-duplicate query message that originates at a node B inside the horizon, or which entered the horizon at node B over its incident edges e if the count $C_{e,B}$ does not exceed a predefined threshold value.

For the horizon criterion to work each query message needs to store the identity of the last k nodes visited. The groups formed by node A in the graph of Fig. 1 according to the horizon criterion are shown in Table 2.

Table 1. Groups formed according to the Hops criterion by node A in the graph of Fig. 1

Hops	1	2	3	4	5	6	7
Groups of nodes formed by node A	B	C	D, J	E, K	F	G, H	I

Table 2. Groups formed according to the Horizon criterion by node A in the graph of Fig. 1

Node in A's horizon	B	C	D	J
Groups of nodes formed by node A	B	C	D, E, F, G, H, I	J, K

The algorithm using the combination of the two criteria described above, namely the horizon+hops, is shown below. For this criterion each message should store the number of hops traversed and the identity of the last k nodes visited.

Feedback-based algorithm using the Horizon+Hops criterion

1. **Warm-up phase**
   - **a. & b.** Same as in Hops criterion
   - **c.** Each node A, for each incident edge e, counts the percentage of duplicates produced on edge e for all queries messages originating at a node B inside A's horizon, or which entered A's horizon at node B and originated k hops away. Let us denote this count by $C_{e,B,k}$.
2. **Execution phase**
   - **a.** Each node A forwards an incoming non-duplicate query message originating at some node B inside A's horizon, or which entered A's horizon at node B and originated k hops away, over its incident edges e if the count $C_{e,B,k}$ does not exceed a predefined threshold.

The groups formed by node A in Fig. 1 for the horizon+hops criterion are shown in Table 3.

We should emphasize that in order to avoid increasing the network traffic due to the feedback messages, a single collective message is returned to each upstream node at the end of the warm-up phase.

Table 3. Groups formed according to the Horizon+Hops criterion by node A in the graph of Fig. 1

Node in A's horizon	B	C	D					J	
Hops	1	2	3	4	5	6	7	3	4
Groups of nodes formed by node A	B	C	D	E	F	G, H	I	J	K

4 Random vs. Small-World Graphs

Two types of graphs have been mainly studied in the context of P2P systems. The first is random graphs which constitute the underline topology in today's commercial P2P systems [7,9]. The second type is small-world graphs which emerged in the modelling of social networks [4]. It has been demonstrated that P2P resource location algorithms could benefit from small-world properties. If the benefit proves to be substantial then the node connection protocol in P2P systems could be modified so that small-world properties are intentionally incorporated in their network topologies.

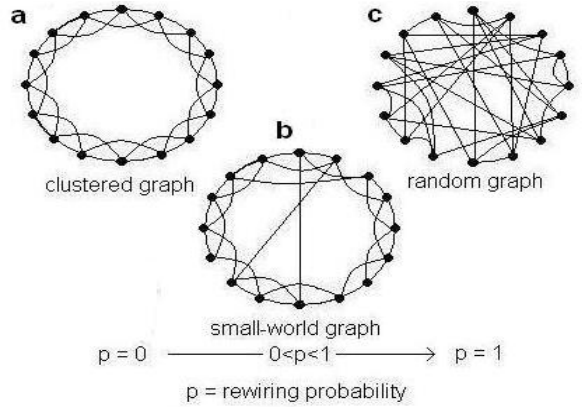


Fig. 2. (a) A clustered graph with no rewired edges (rewiring probability p=0). (b) A small-world graph produced from the clustered graph with a small rewiring probability (c) A random graph produced if every edge is rewired to a random node (rewiring probability p=1)

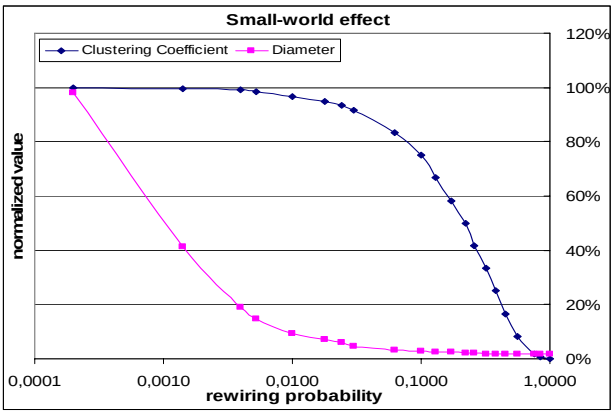


Fig. 3. By rewiring a few edges of the initial clustered graph to random nodes the *average diameter* of the graph is greatly reduced, without significantly affecting the *clustering coefficient*

In *random graphs* each node is randomly connected to a number of other nodes equal to its degree. Random graphs have small diameter and average diameter. The *diameter* of a graph is the length (number of hops for un-weighted graphs) of the longest among the shortest paths that connect any pair

of nodes. The *average diameter* of a graph is the average of all longest shortest paths from any node to any other node.

A *clustered graph* is a graph that contains densely connected "neighborhoods" of nodes, while nodes that lie in different neighborhoods are more loosely connected. A metric that captures the degree of clustering that graphs exhibit is the clustering coefficient. Given a graph G, the *clustering coefficient of a node* A of G is defined as the ratio of the number of edges that exist between the neighbors of A over the maximum number of edges that can exist between its neighbors (which equals k(k-1) for k neighbors). The *clustering coefficient of a graph* G is the average of the clustering coefficients of all its nodes. Clustered graphs have, in general, higher diameter and average diameter than their random counterparts with about the same number of nodes and degree.

A small-world graph is a graph with high clustering coefficient yet low average diameter. The small-world graphs we use in our experiments are constructed according to the Strogatz-Watts model. Initially, a regular, clustered graph of N nodes is constructed as follows: each node is assigned a unique identifier from 0 to N-1. Two nodes are connected if their identity difference is less than or equal to k (in mod N arithmetic). In Fig. 2(a) such a graph is shown for N=16 and k=2. Subsequently, each edge of the graph is rewired to a random node according to a given rewiring probability p. If the rewiring probability of edges is relatively small, a small-world graph is produced (high clustering coefficient and small average diameter), as shown in Fig. 2(b). As the rewiring probability increases the graph becomes more random (the clustering coefficient decreases). For rewiring probability p=1, all graph edges are rewired to random nodes, and this results to a random graph, Fig. 2(c). In Fig. 3, we can see how the clustering coefficient and the average diameter of graphs vary as the rewiring probability p increases. Small-world graphs are somewhere in the middle of the x axis (p=0.01).

The clustering coefficient of each graph is normalized with the respect to the maximum clustering coefficient of a graph with the same number of nodes and average degree. In what follows, when we refer to the clustering coefficient of a graph with N nodes and average degree d, denoted by CC, we refer to the percentage of its clustering coefficient over the maximum clustering coefficient of a graph with the same number of nodes and average degree. The maximum clustering coefficient of a graph with N nodes and average degree d is the clustering coefficient of the clustered graph defined according to the Strogatz-Watts model, Fig. 2(a), before any edge rewiring takes place.

Fig. 4 shows the percentage of duplicates messages generated per hop over the messages generated on that hop on a random and on a small-world graph of 2000 nodes and average degree 6. We can see from this figure that in a random graph there are very few duplicate messages in the first few hops (1-4), while almost all messages in the last hops (6-7) are duplicates. On the contrary, in small-world graphs duplicate messages appear from the first hops and their percentage (over the total number of messages per hop) remains almost constant till the last hops.

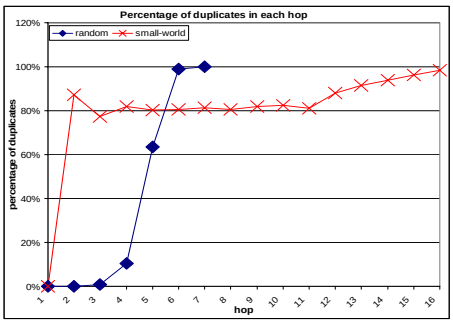


Fig. 4. Percentage of messages generated per *hop*, which are *duplicates*, in random and small-world graphs. In small-world graphs the percentage of duplicates in hops 2 to 11 is almost constant, while in random graphs, in small hops there are no duplicates and in large hops almost all messages are duplicates

5 Experimental Results on Static Graphs

The simulation was performed using **sP2Ps** (simple P2P simulator) developed at our lab. The experiments were conducted on graphs with 2000 nodes and average degree 6. The clustering coefficient ranged from 0.0001 to 0.6, which is the maximum clustering coefficient of a graph with N=2000 and d=6. We shall refer to CC values from now on, as percentages of that max value. We conducted experiments for different values of the algorithm's parameters. The horizon value varied from 0 (were practically the horizon criterion is not used) up to the diameter of the graph. Furthermore, we used two different threshold values, namely 75% and 100%, to select the connections over which

messages are forwarded. For example a threshold of 75% indicates that if the percentage of duplicates on an edge e during the warm up phase exceeds 75% for messages originated at the nodes of a group, in the execution phase no query message from this group is forwarded over edge e. The TTL value is set to the diameter of the graph.

The efficiency of our algorithm is evaluated based on two metrics, firstly the percentage of duplicates sent by the algorithm, in relation to the naive flooding and secondly the network coverage (defined as the percentage of network nodes reached by the query). Thus, the lower the duplicates percentage and the higher the coverage percentage, the better. Notice that a threshold value of 100% indicates that messages originating at the nodes of a group are not forwarded only over edges that produce exclusively (100%) duplicates for all nodes of that group during the warm-up phase. In this case we do not experience any loss in network coverage but the efficiency of the algorithm in duplicate elimination could be limited. In all experiments on static graphs, the warm-up phase included one flooding from each node. In the execution phase, during which the feedback-based algorithm is applied, again one flooding is performed from each node in order to gather the results of the simulation experiment.

In Figs 5-10 we can see the experimental results for the feedback-based algorithm with the horizon criterion. In Fig. 5 we can see the percentage of duplicates produced as a function of the percentage of graph nodes in the horizon for three graphs (random with CC=0.16, clustered with CC=50, and small-world with CC=91.6) and for threshold value 100%, which means that there is no loss in network coverage. We can deduce from this figure that the efficiency of this algorithm is high for clustered graphs and increases with the percentage of graph nodes in the horizon. Notice that in clustered graphs, with a small horizon value a larger percentage of the graph is in the horizon as compared to random graphs. In Fig. 6 we plot the percentage of duplicates produced by the algorithm as a function of the clustering coefficient for horizon value 1 and threshold 100%. We can see that even for such a small horizon value the efficiency of the algorithm increases linearly with the clustering coefficient of the graph. We can thus conclude that the feedback-based algorithm with the horizon criterion is efficient for clustered and small-world graphs.

Even if the percentage of graph nodes in the horizon decreases, in case the graph size increases and the horizon value remains constant, the efficiency of the algorithm will remain unchanged, because in clustered graphs the clustering coefficient does change significantly with the graph size. Thus, the horizon criterion is scalable for clustered graphs. In contrast, in random graph, in order to maintain the same efficiency as the graph size increases, one would need to increase the horizon value, in order to maintain the same percentage of graph nodes in the horizon. Thus the horizon criterion is not scalable on random graphs.

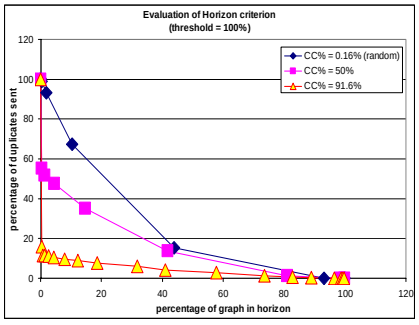


Fig. 5. Percentage of *duplicates* as a function of the percentage of graph *nodes in the horizon* for three graphs with clustering coefficients 0.16, 50, and 91.6, and threshold value 100%

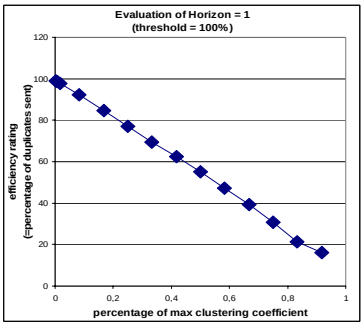


Fig. 6. Percentage of *duplicates* as a function of the *clustering coefficient* for horizon value 1 and threshold value 100%

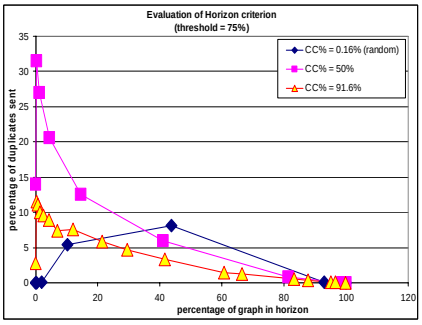


Fig. 7. Percentage of *duplicates* as a function of the percentage of graph *nodes in the horizon* for three graphs with different clustering coefficients (0.16, 50, and 91.6) and threshold value 75%

Figs 7-10 show the efficiency of the algorithm with the horizon criterion in duplicate elimination for threshold 75%. In Figs 7 and 8 we can see that the algorithm is very efficient on clustered graphs. From the same figures we can see that with this threshold value in random graphs (CC=0.16) most duplicate messages are eliminated but there is loss in network coverage. Thus, even if we lower the threshold value, the horizon criterion does not work well for random graphs. The algorithm's behavior is summarized in Fig. 9, where duplicate elimination, denoted by D, and network coverage, denoted by C, are combined into one simple metric, defined as C^2D.

In Fig. 10 we can see again the efficiency of the algorithm for horizon value 1 (as in Fig. 6) but for a threshold of 75%. Notice that the algorithm's efficiency is not linear to the percentage of the clustering

coefficient of the graph. This arises because the threshold value of 75% is not necessarily the best choice for any clustering coefficient.

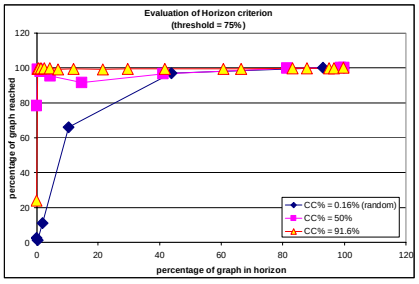


Fig. 8. Network *coverage* as a function of the percentage of graph *nodes in the horizon* for three graphs with clustering coefficients 0.16, 50, and 91.6 and threshold 75%

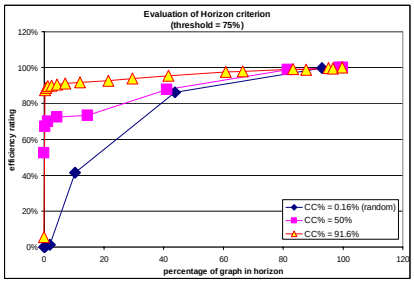


Fig. 9. *Efficiency* of the feedback based algorithm as a function of the percentage of graph *nodes in the horizon* for three graphs with clustering coefficients 0.16, 50, and 91.6 and threshold 75%

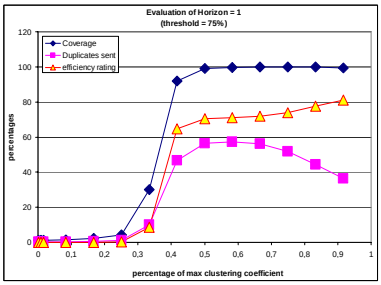


Fig. 10. Network *coverage*, percentage of *duplicates*, and *efficiency* as a function of the clustering coefficient for horizon value 1 and threshold 75%

In Fig. 11 we can see the experimental results for the algorithm with the hops criterion for a graph with 2000 nodes and average degree 6 while varying the clustering coefficient. We can see in this figure that the hops criterion is very efficient in duplicate elimination, while maintaining high network coverage, for graphs with small clustering coefficient. This means that this criterion exhibits very good behaviour on random graphs. As the clustering coefficient increases the performance of the algorithm with the hops criterion decreases. This behaviour can be easily explained from Fig. 4, where the percentage of duplicates per hop is plotted for random and small-world graphs. We can see from this figure that in random graphs, the small hops produce very few duplicates, while large hops produce too many. Thus, based on the hops criterion only, we were able to eliminate a large percentage of duplicates without greatly sacrificing network coverage.

As mentioned before, the hops criterion works better for random graphs. In case the graph size increases, the number of hops also increases (recall that the diameter of a random graph with N nodes and average degree d is log(N)/log(d)). Thus, the hops criterion is scalable on random graphs.

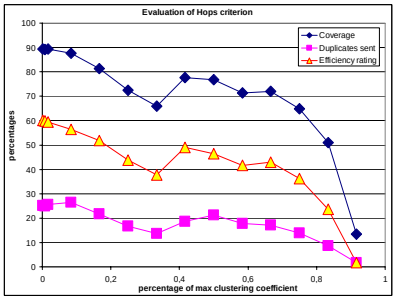


Fig. 11. Network *coverage*, percentage of *duplicates*, and *efficiency* of the algorithm with the hops criterion as a function of the clustering coefficient

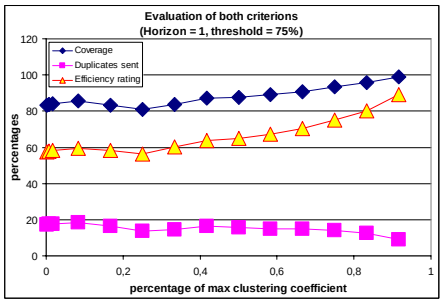


Fig. 12. *Network coverage*, percentage of *duplicates*, and *efficiency* of the algorithm with the horizon+hops criterion as a function of the clustering coefficient

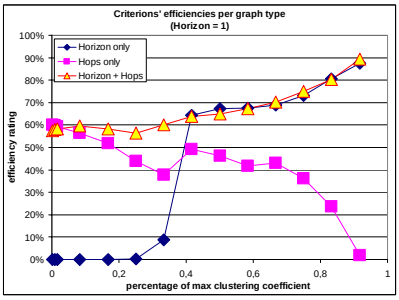


Fig. 13. *Efficiency* of algorithms with the *horizon*, *hops*, and *horizon+hops* criteria as a function of the clustering coefficient and for horizon value 1

In Fig. 12, we see the efficiency of the algorithm for the horizon+hops criterion. As we can see from this figure this combination of criteria constitutes the feedback based algorithm efficient in graphs with all clustering coefficients, random and small-world. In Fig. 12, three different metrics are plotted, the network coverage, the percentage of duplicates, and the efficiency as a function of the clustering coefficient of the graph. We can see that for any clustering coefficient network coverage is always above 80%, while the percentage of duplicate messages not eliminated is always less than 20%. This behavior is achieved for random and small-world graphs for horizon value of only 1. Thus the horizon+hops criterion is scalable on all types of graphs.

In Fig. 13 we compare the efficiencies of the hops, horizon, and horizon+hops and we see that their combination, horizon+hops works better than each criterion separately.

6 Experimental Results on Dynamic Graphs

In what follows, we introduce dynamic changes to the graph, meaning that a graph node can leave and some other node can enter the graph, and we monitor how these changes influence the algorithm's efficiency. We introduced a new parameter to our experiments in order to capture the rate of graph change. This parameter measures in query-floods the lifetime of a node in the graph. A graph rate change of r means that each node will initiate, on the average, r query-floods before leaving the network. Insertion of new nodes is performed so as to preserve the clustering coefficient of the graph.

We also introduce a dynamic way to determine when the warm-up phase can terminate, meaning that we have collected enough measurements. The warm-up phase for a group of nodes terminates after the percentage of duplicates seen on an edge for messages originating at nodes of the group stops to oscillate significantly. More specifically, the warm-up phase terminates on an edge for a group of nodes, if in each of the last 20 rounds the change in the count (percentage of the number of duplicates seen on that edge for messages originating at nodes of the that group) was smaller that 2% and the total change over the last 20 rounds was smaller that 1%.

We perform experiments for random graphs and for small-world graphs with clustering coefficient CC=33 and CC=84. For each of these graphs, the value of the change rate equals 0 (static graph), 1, 50, and 200. A change rate of 200 indicates that each node will make 200 query-floods before leaving the network, which is a reasonable assumption for Gnutella 2 [7]. This is because each Ultrapeer contains, on the average, 30 leaves. A leaf node has in general much smaller average lifetime than an Ultrapeer, which means that each Ultrapeer will "see" more than 30 unique leaves in its lifetime. If we assume that each leaf node will send one query through the Ultrapeer, this explains the fact that real-world measures with an Ultrapeer show that each Ultrapeer sends about 100 queries per hour. For each of these graphs and change rates, we run experiments with the following Horizon values:

- Horizon values = {1|2} for random graphs and for small-world graphs with CC = 33.
- Horizon values = {1|4} for small-world graphs with CC = 84.

We performed two experiments with the same horizon value, one using the hops criterion and one without the hops criterion. The threshold value was set to 75%. Each experiment performed 25*2000 floods. The difference between the values "0 wo act. threshold" and "0 with act. threshold" in the x axis in the figures indicates that in both cases the change rate is 0 (static graph), but in the first case, the numbers are taken from the experiments described in the previous section, while in the second case the activation threshold was used to terminate the warm-up phase. This enables us to clearly see the benefit of the activation threshold.

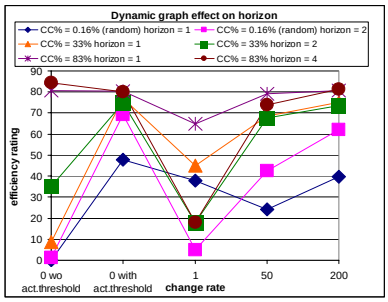


Fig. 14. Performance (efficiency) of the algorithm on a dynamic graph for the horizon criterion

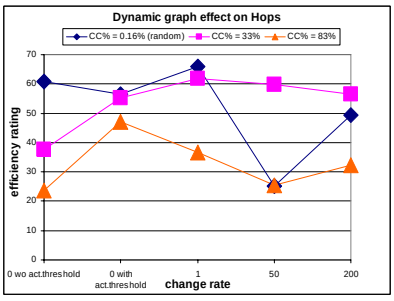


Fig. 15. Performance (efficiency) of the algorithm on a dynamic graph for the hops criterion

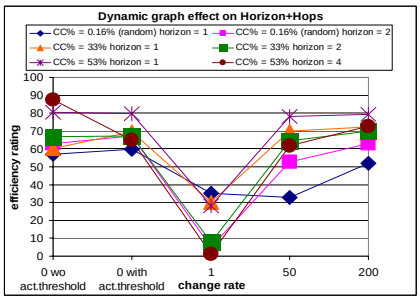


Fig. 16. Performance (efficiency) of the algorithm on a dynamic graph for the horizon + hops criterion

Fig. 14 shows how the algorithm performs on dynamic graphs for the horizon criterion. We should first note that the use of the activation threshold increases the efficiency of the algorithm significantly. This happens because nodes gradually start eliminating traffic for certain groups of nodes instead of all of them starting eliminating duplicates for all groups simultaneously.

We can see that the efficiency of the algorithm decreases when the change rate is 1. The main reason for this is not that the measurements for each group quickly become stale, but rather because each node needs some warm-up period to learn the topology of the network. A certain amount of traffic needs to be "seen" by any node, to make the necessary measurements. If that time is a large fraction of the node's lifetime, it means that it will spend most of its time measuring instead of regulating traffic according to the measurements.

Finally and most importantly, we can see that the results for a change rate of 200 are the same as those of a change rate of 0 with activation threshold, which shows that, given that the warm-up phase is

shorter than the time during which the nodes use the algorithm (execution phase), the changes of the graph do not affect the algorithm's efficiency.

In Fig. 15 we can see that the activation threshold is beneficial to the algorithm with the hops criterion. Furthermore, from the same figure, it becomes clear that the efficiency of the feedback-based algorithm with the hops criterion is not greatly affected by the dynamic changes in the graph. We should however point out that it seems to lightly affect the efficiency of the algorithm in highly clustered graphs.

In Fig. 16 we finally see the efficiency of the algorithm for the horizon+hops criterion. We should notice again that the use of the activation threshold does not harm the algorithm, except in the case of the graph with high clustering coefficient and for a horizon value greater than 1. However, as we have seen before, there is not reason to use a horizon value larger than 1. Again, the change rate does not affect the measurements for groups of nodes, since the reason for the low efficiency at high change rates is the fact that the nodes spent most of their lifetime in the warm-up phase.

7 Conclusions

We presented the feedback-based algorithm, an innovative method which reduces significantly the number of duplicate messages produced by flooding while maintaining high network coverage. The algorithm monitors the percentage of duplicates on each connection during a warm-up phase, and directs traffic to connections that do not produce excessive number of duplicates during the execution phase. In order for this approach to work, each network node groups together the rest of the nodes according to some criteria, so that nodes that produce many duplicates on its incident edges are in different groups than those that produce only few duplicates. The efficiency of the algorithm was demonstrated through extensive simulation on random and small-world graphs, the two most common types of graphs that have been studied in the context of P2P systems. The experiments involved graphs of 2000 nodes. The feedback-based algorithm was shown to reduce to less than 20% the number of duplicates of flooding while conserving network coverage above 80%. The memory requirements in each node are much less compared to the algorithm that constructs shortest paths trees from each network node. The efficiency of our algorithm was demonstrated on static and dynamic graphs.

References

1. Y. Chawathe, S. Ratnasamy, and L. Breslau. *Making Gnutella-like P2P Systems Scalable*. ACM SIGCOMM, 2003.
2. Crespo and H. Garcia-Molina. *Routing Indices for Peer-to-Peer Systems*. International Conference on Distributed Computing Systems, 2002.
3. Crespo and H. Garcia-Molina. Semantic Overlay Networks for P2P Systems. 2002.
4. Duncan, J. Watts, and S. H. Strongatz. *Collective Dynamics of Small-world Networks*. Nature, Vol. 393, pp. 440-442, 1998.
5. C. Gkantsidis, M. Mihail, and A.Saberi. *Hybrid Search Schemes for Unstructured Peer-to-Peer Networks*. IEEE INFOCOM, 2005.
6. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, *Search and Replication in Unstructured Peer-to-Peer Networks*. International ACM Conference on Supercomputing, 2002.
7. R. Manfredi and T. Klingberg. Gnutella 0.6 Specification, http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html
8. M. Ripenau, I. Foster, A. Iamnitchi, and A. Rogers. *UMM: A Dynamically Adaptive, Unstructured, Multicast Overlay*. In Service Management and Self-Organization in IP-based Networks, 2005, editors M. Bossardt, G. Carle, D. Hutchison, H. de Meer, and B. Plattner, Dagstuhl Seminar Proceedings.
9. Sharman Industries. Kazaa, http://www.kazaa.com
10. K. Sripanidkulchai, B. Maggs, and H. Zhang, Efficient Content Location using Interest-Based Locality in Peer-to-Peer Systems. IEEE INFOCOM, 2003.
11. D. Stutzbach and R. Rejaie. *Characterizing Today's Gnutella Topology*. Technical Report CIS-TR-04-02, Department of Computer Science, University of Oregon, Dec. 2004.
12. D. Tsoumakos and N. Roussopoulos. *A Comparison of Peer-to-Peer Search Methods*. International Workshop on the Web and Databases, 2003.
13. Z. Zhuang, Y. Liu, L. Xiao, and L.M. Ni. *Hybrid Periodical Flooding in Unstructured Peer-to-Peer Networks*. International Conference on Parallel Computing, 2003.
14. D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. *Exploiting Locality for Scalable Information Retrieval in Peer-to-Peer Systems*. Information Systems Journal, Vol. 30, No. 4, pp. 277-298, 2005.

User Management for Virtual Organizations

Jiří Denemark[3], Michał Jankowski[2], Luděk Matyska[13],
Norbert Meyer[2], Miroslav Ruda[1], and Paweł Wolniewicz[2]

[1] Institute of Computer Science, Masaryk University, Botanická 68a,
602 00 Brno, Czech Republic
{ludek, ruda}@ics.muni.cz
[2] Poznań Supercomputing and Networking Center,ul. Noskowskiego 10,
61-704 Poznań, Poland
{jankowsk,meyer,pawelw}@man.poznan.pl
[3] Faculty of Informatics, Masaryk University, Botanická 68a,
602 00 Brno, Czech Republic
jirka@ics.muni.cz

Abstract. Scalable and fine-grained Grid authorization requires the move away from grid-mapfile based access control and 1-to-1 mappings to individual OS user accounts. This is recognized and addressed by virtual organization (VO) authorization services, e. g. VOMS/LCAS and CAS. They do, however, not address user OS account management and isolation/sandboxing requirements, such as flexible pooling of accounts while maintaining auditing records. This paper describes some existing systems for user management for VOs and provides a list of requirements for a new user management system which our current research is focused on.

1 Introduction

The main aim of user management system is controlled, secure access to grid resources. Security requires authentication of the user and authorization based on combined security policy from the resource provider and virtual organization of the user. The second important thing is possibility of logging user activities for accounting and auditing and then gathering these data both by the resource provider and virtual organization of the user. From the user point of view, an important feature is single sign-on.

The problem of user management is a non-trivial one in an environment, that includes bulk number of computing resources, data, and hundreds or even thousands of users participating in lots of virtual organizations. The complexity rises from the point of view of time required for administration tasks and automation of these tasks. There are many solutions that attempt to fulfill these basic requirements and solve the mentioned problem, but none of them, according to our best knowledge, solve the problem in complex and satisfactory way.

2 Definitions

Virtual organization (VO) is a set of individuals and/or institutions that allows its members sharing resources in a controlled manner, so that they may collaborate to achieve a shared goal [1].

We assume that virtual organizations may form hierarchies. The hierarchy of VO is useful for user management on the VO side (delegation of administrative burden to sub-organization in case of big organizations) and accounting (sub-organizations may refer to real institutions and departments who are responsible for paying the bills). The hierarchy forms a Directed Acyclic Graph (DAG) where the VOs are vertices and the edges represent relations between them (see [3], sub-organizations are called "groups").

The user may be a member of many VOs, and in particular, member of a sub-organization is also member of parent organization.

The privileges the organization wants to grant the user, related to the tasks he is supposed to perform, are connected to *user roles*. The roles are defined across the hierarchy of VOs and are managed in independent structure, although the authorities of VOs are responsible for defining roles. One user may have multiple roles and he is responsible to select the required role while accessing the resource.

Any special rights to resources expressed, e. g., by ACL [2] are called *capabilities*. The capabilities may be used to express any rights to a specific user, e. g., some file is writable only by the owner.

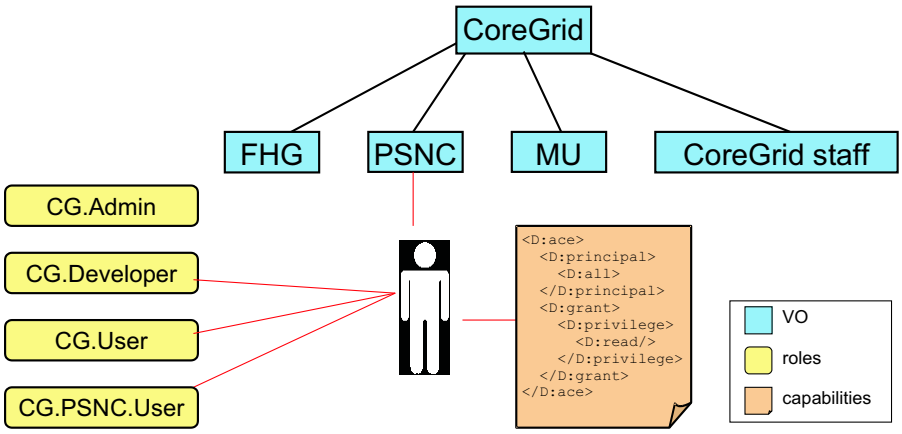


Fig. 1. Hierarchy of Virtual Organizations, User Roles and Capabilities

Resource provider (RP) is an abstract entity that owns and offers some resources (e. g. services, CPU, disks, data, etc.) to the grid users.

By the *virtual environment* we understand encapsulation of user jobs in order to give it a limited set of privileges and be able to identify the user and organization on behalf which the job acts. Example implementations are virtual accounts [8], virtual machines, and sandboxes [5].

3 Existing Solutions

In this section, we provide a brief description of several systems trying to cope with user management in the context of virtual organizations.

3.1 Perun

Perun [9] provides a repository of complex authorization data, as well as tools to manage the data. The data are used to generate configuration of the authorization services themselves (starting from UNIX user accounts throught grid-mapfiles to VOMS database). In turn, these services are used to enforce authorization policies.

Perun makes use of central configuration repository which models an *ideal world*, i. e. how the resources should look like. In this central repository, all the necessary (and possibly very complex) integrity constraints are relatively easy to be enforced. The repository is complemented with a change propagation mechanism which detects the changes, generates consistent configuration snapshots of atomic pieces of managed systems, and tries to deliver them to their final destinations, dealing with resource or network failures appropriately. In this way, the *real world* is forced to follow the ideal one as closely as possible.

The core of the system is completely independent on the structure and semantics of the configuration data, hence the system is easily extensible.

3.2 Virtual User System

Virtual User System (VUS) [8] is an extension of the system that runs users' jobs (e. g. scheduling system, Globus Gatekeeper, etc.) and allows running jobs without having a personal user account on a node. The personal accounts are replaced by "virtual" ones, that are mapped to users only for time needed to fully process a job.

There are groups of virtual accounts on each computing node. Each group consists of accounts with different privileges, so the fine grain authorization is achieved by selecting appropriate group by an authorization module. The authorization module is pluggable and may be easily extended or replaced. For example, the authorization decision may be based on VO-membership of the user and checking banned user list. The mapping user-account mechanism assures that only one user is mapped to a particular account at any given time. The history of user-account mappings is stored in a database, so that accounting and tracking user activities are possible.

3.3 VOMS, LCAS and LCMAPS

Virtual Organization Membership Service (VOMS) [2] contains database with information on the user's Virtual Organization and group (sub-organization) membership, roles and capabilities. The service preserves it in a special format—the VOMS credential. The user, before starting a job must acquire the VOMS

proxy certificate signed by his VO and valid for limited time. The extra authorization data is placed as a non-critical extension in the proxy, so it is compatible with not VOMS aware services.

In order to take an advantages of VOMS data, the Globus Gatekeeper was extended by LCAS and LCMAPS. Local Center Authorization System (LCAS) is a service used on computing nodes in order to enforce local security policies. Local Credential Mapping Service (LCMAPS) maps user to local credentials (AFS/Kerberos tokens, UNIX account and group), depending on user proxy certificate and job description.

3.4 Virtual Workspaces, Runtime Environments, Dynamic Virtual Environments

Very interesting work in the area was done by researchers from Argonne National Lab and University of California [4–6]. They proposed and implemented Workspace Management Service, which allows to run user jobs in virtual environment (see section 2), using different technologies (GRAM Gatekeeper, OGSI, WSRF). The virtual environments are implemented as dynamically created Unix accounts and virtual machines.

These works deal widely with problems connected to job encapsulation and provide some efficiency comparisons of different virtual environment implementations based on tests performed on prototype system and testbed. The authorization issues are addressed closer only by [4], where RSL-based policy language was proposed.

4 System Requirements

4.1 Authentication

The first step in obtaining an access to a remote resource is authentication. From the user point of view, the remote access should be as simple as possible and similar to the local access. This may be achieved by features like single sign-on, delegation, integration with local security solutions, user based trust relationships (resources from different providers may be used together by the user without need of any security configurations done amongst these RPs) [1]. The mentioned requirements are fulfilled by Globus Toolkit [10] to a great extent.

4.2 Authorization

The concept of virtual organizations allows for easier decentralization of user management (each VO is responsible for managing some group of users). On the contrary, in the classical solution each computing node must store user auhorisation information locally (e. g. in Globus grid-mapfile), which obviously is not scalable and brings a nightmare of synchronization. On the other hand, the resource provider must have full control on who and how uses his resources. So that, the security policy should be combined from two sources: VO and RP.

The second important issue is fine grained authorization [4], that allows limiting user access rights to specific resources. The authorization is based on the triplet VO, role, capabilities [2] and is done on the computing node. The RP policy defines privileges for given pair VO-role and interprets the capabilities. RP policy may limit the privileges in any way, including denying access at all. The virtual environment should be able to enforce the (limited) privileges.

User's request should be self-contained so that RP does not need to contact any external entity to obtain any information (such as VO, role(s), capabilities) required to authorize the user. This additional information must be stored within the request in an expandable way.

The authorization module should be plug-in based in order to allow flexible configuration (use different set of plug-ins with different priorities) and easy integration with existing authorization systems, services or mechanisms.

In some applications, (e. g. pre-paid systems) it may be required to suspend or delete a job after quotas expiration/overdraw. Related problem is estimation of resource usage before starting the job in order to avoid canceling jobs done in 90 %. The quota should be soft or it should be possible to suspend the job for some time.

4.3 Encapsulation of Jobs and Results

The system must assure, that two different jobs will not interact in unwanted and unpredictable way, e. g. overwriting each other results. Moreover, it must be possible to identify who and when used specific resources, performed or attempted some actions. This is relevant from the security and accounting point of view. Usually it is not even enough to know the identity of a user, but it is important on whose (which VO) behalf he acted and in which role.

The basic model (default configuration) is that all jobs are encapsulated and thus isolated one to another. Final or partial results of a job are not available to other jobs until they are stored to the global space. However, in some situations (e. g. workflows of jobs that may share temporary files) some cooperation of jobs or access to the same resources may be required. It should be possible to specify if the job should run in an existing environment or in a new one. Possibly, system should detect such situation and handle it automatically.

In complex task, it may be required that the user may have to use multiple roles or identities to gain access to multiple resources. It should be possible to separate subtasks preformed with different privileges, identities (certificates) and on behalf of different VOs.

Files stored with some local privileges/IDs/tokens, may be accessed later with different ID (certificate), with different local ID/token. Special access privileges may be required for VO authorities in order to control or even stop the user's jobs and to check accounting data. Access to virtual environment for other users, pointed by the user, may be required for interactive jobs that require cooperation.

Possibly, job encapsulation should provide a secure execution environment for jobs, in which no-one except authorized persons should be able to read and interpret the input and output data, even the RP.

4.4 Accounting and Logging Facilities

Any production grid, especially commercial one, needs accounting feature. The accounting data must be stored with a proper context (user, VO, role, capabilities, time) and them collected (possibly from several locations) by users, VOs and RPs. In many application the standard system mechanisms (such as e. g. Unix accounting) offer enough information, but the system should be also capable of storing non standard accounting data.

From the security point of view, it is important to track user activities (e. g. by analyzing system logs) in order to detect any rule-breaking. It must be possible to identify the user who has performed a particular action.

Both of the above require proper encapsulation of jobs (as described in the previous section) and storing history of user-virtual environment mappings.

4.5 Other Requirements

There are also several other requirements, that the user management system should met.

It should be possible to combine "classical" and "virtual" user management in some system.

The system architecture must be ready for lightweight virtual organizations (very dynamically created and removed after short time). This high dynamics shall not introduce much administrative burden or computational overhead.

Granting rights to the "long-lived" resources (like files) for users that changed certificates should be handled. CAs should track the history of issued certificates and associated physical users (i. e. proper *identity* management is necessary).

The architecture should be modular and flexible. It should give a chance for easy integration with existing solutions and standards. The modularity embraces plug-in based authorization, replaceable virtual environment module (that allows different implementations of VEs).

Automatic registering of new users, issuing certificates and any special security considerations connected with this may be required in some solutions (e. g. application-on-demand).

5 Proposed Solution

We realized, while analyzing existing solutions, that there are number of tools that provide for at least part of the functionality mentioned in section 4, however none addresses all the issues. These tools are widely used in number of projects and some of them become kind of standard, so it seems to be reasonable to be compatible with them. Moreover, it makes no sense to implement from scratch an existing functionality. So we propose to put them into pluggable framework, that will combine the features gaining the synergy effect.

The other important design assumption is being concordant to the existing standards and trends in the area of grid computing, especially webservice (WS)

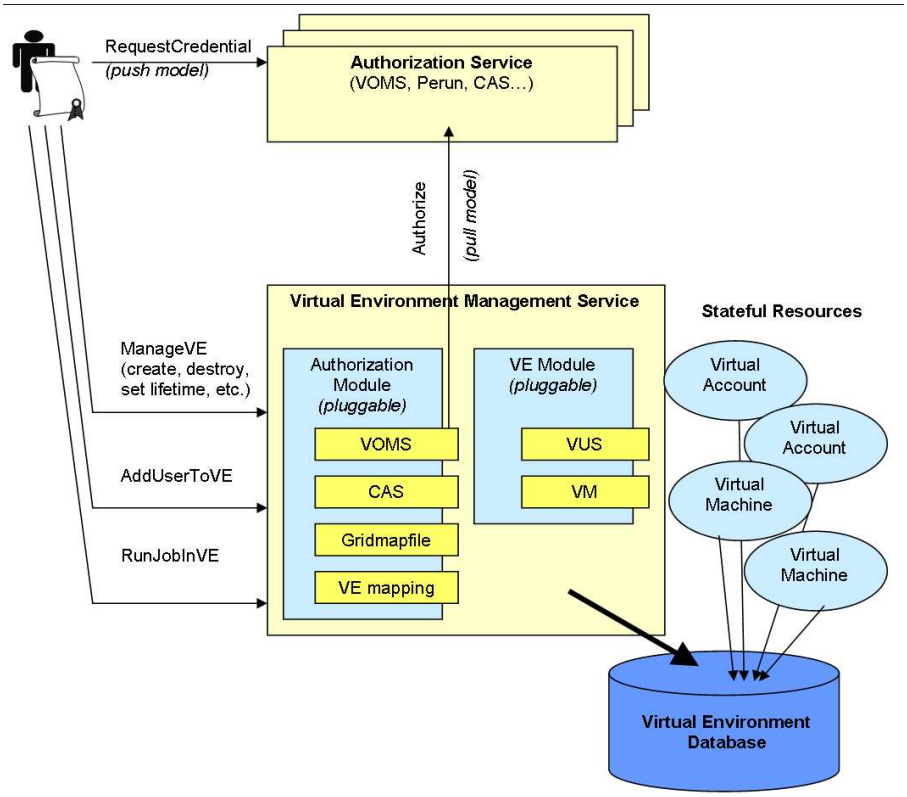


Fig. 2. Virtual Environment Management Service

approach. WS-Stateful Resource [7] technology seems to be especially promising for our purpose, as it allows for easy modeling virtual environment and managing its life cycle.

We propose webservice responsible for managing virtual environments (Virtual Environment Management Service, Fig. 2), especially creating and destroying them as well as running jobs in them. In the background, the service collects data on the virtual environments concerning time of creation and destruction, users mapped to the environment, accounting and logging information. These data will be available to different players on the scene like the users, managers of VOs, resource owners etc. via the second webservice called Virtual Environment Information Service, Fig. 3.

5.1 Virtual Environment Management Service

The Virtual Environment Management Service consists of two main modules: authorization module and virtual environment module.

The authorization module performs authentication first. This may be based on existing Globus GSI. The authorization is done by querying set of authorization plugins. The set is configurable, so that the administrator may tune local authorization policy to the real Grid needs and abilities. We plan to implement plugins for most often used authorization mechanisms and services like gridmapfile (possibly dynamically updated by Perun), CAS, VOMS. The plugins

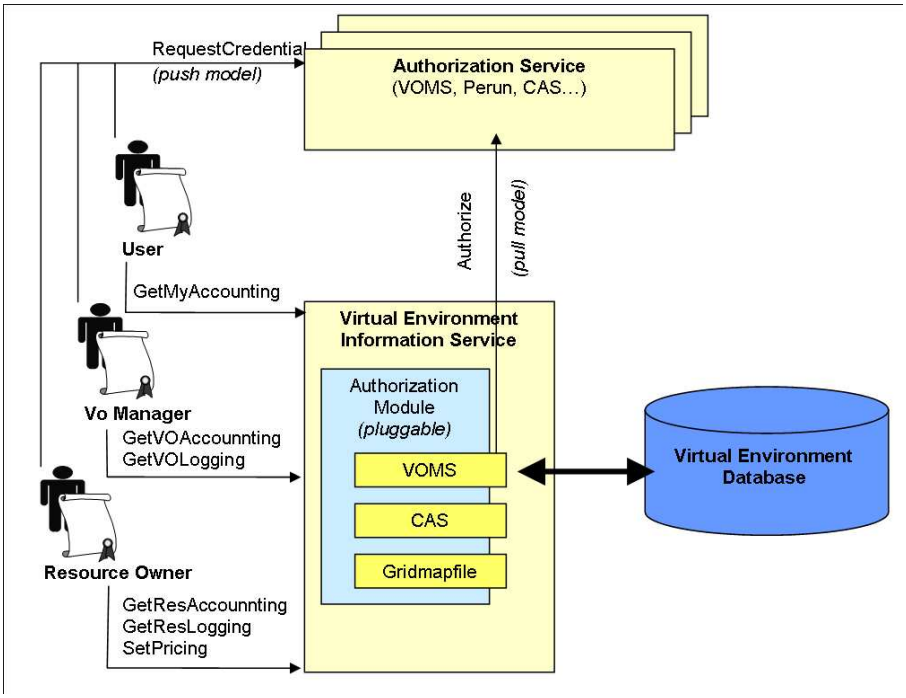


Fig. 3. Virtual Environment Information Service

play role of policy decision points (PDP). The authorization decision itself may be done locally (in case of push model of authorization which we prefer) or by querying remote authorization service (pull model—not preferred, but possible). The plugins should not only answer the question if the user is allowed to perform the specified service action, but also give some hints for the parameters of the virtual environment, e. g. grid-mapfile plugin will tell the (virtual) account name, VOMS will expose ACL that will help with creation of virtual machine with specific limitation.

The special authorization plugin is VE mapping, introduced in order to satisfy requirements about loosing isolation level of the VE. This plugin will check a special Access Control List for the VE. While the VE is created, the list will contain one user—owner (creator) of the VE—with full rights to the VE. Then, the owner can add users to the VE either with limited (e. g. only read access to files or job execution) or full (including VE life cycle management) privileges. Note, that the user—owner of VE—takes much responsibility for the added users, because while loosing isolation level we also loose requirements for identifying user and context. The extent of this loosing depends on VE implementation; namely it is impossible to distinguish users if they run jobs in the same virtual account, but it is possible to distinguish them if they run jobs on different accounts of the same virtual machine.

Virtual environment module is responsible for creation, deletion and communication with virtual environments, implemented as stateful resources. The module is also pluggable, so it is possible to use different implementations of VE. It is planned to implement Virtual User System plugin and at least one plugin

for virtual machine. The module records all its relevant operations (especially like VE creation and deletion) to the Virtual Environment Database.

5.2 Virtual Environment Database

The records of VE operations together with the standard system logs and accounting data will provide complete information on user actions and resource usage. However these two sources must be combined and the result put to the database. This might be implemented as database trigger that collects the logging and accounting data periodically or while the VE is destroyed. It must be also possible to put some non-standard data to the database (e. g. time of usage of laboratory equipment connected to the system).

For billing purposes accounting information must be connected with prices. The price is computed depending on the pricing policy of the resource owner. In the simplest model, the database contains a dynamic price list and the current price is put together with accounting record.

5.3 Virtual Environment Information Service

This service is a front-end for the Virtual Environment Database. The access to the data must be authorized and depends on the user role:

- *Common users*, who have run jobs—they should have rights to read the accounting data referring to themselves (e. g. in order to control their budget).
- *Managers of virtual organizations*—should be able to read logging and accounting data of all VO members, to have full control on budget and behavior of the users.
- *Owners of resources*—should be able to access all the data connected to the resource. In the simplest case, the resource is the whole local node or cluster on which the VE runs and the owner is the system administrator. In more sophisticated case the resources may be differentiated and there can be more owners (e. g. usage of some software, owed by some local user may be subject for the accounting). The owners also should have right to modify the pricing policy.

This service will require authorization module with set of plugins similar to the Virtual Environment Management Service, but they must take a bit different decision, mainly based on the user role mentioned above.

6 Summary

In the paper we discussed in detail requirements for user management in Grid environment with a special respect to the Virtual Organization concept. Examples of the existing approaches to the problem were briefly described. As none of the existing approaches address all the requirements, but most of the requirements are addressed by some approach, we propose a system that will be a framework for the existing solutions, allowing combination of their features.

7 Acknowledgment

This work has been supported by the CESNET Research Intent (MSM6383917201) and by the EU CoreGRID NoE (FP6-004265).

References

1. I.Foster, C.Kesselman, S.Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, International J. Supercomputer Applications, 15(3), 2001.
2. R.Alfieri, R.Cecchini, V.Ciaschini, L.Dell'Agnello, A.Frohner, A.Gianoli, K.Lentey, F.Spataro, VOMS: an Authorization System for Virtual Organizations, 1st European Across Grids Conference, Santiago de Compostela, February 13-14, 2003.
3. R.Alfieri, R.Cecchini, V.Ciaschini, L.dell'Angelo, A.Gianoli, F.Spataro, F.Bonnassieux, P.Broadfoot, G.Lowe, L.Cornwall, J.Jensen, D.Kelsey, A.Frohner, D.L.Groep, W.Som de Cerff, M.Steenbakkers, G.Venekamp, D.Kouril, A.McNab, O.Mulmo, M.Silander, J.Hahkala, K.Lorentey Managing Dynamic User Communities in a Grid of Autonomous Resources, Computing in High Energy and Nuclear Phisics, La Jolla, California, 24-28 March 2003.
4. K.Keahey, V.Welch, S.Lang, B.Liu, S.Meder Fine-Grain Authorization Policies in the GRID: Design and Implementation 1st International Workshop on Middleware for Grid Computing, 2003.
5. K.Keahey, K Doering, I.Foster, From Sandbox to Playground: Dynamic Virtual Environments in the Grid, 5th International Workshop in Grid Computing (Grid 2004), Pittsburgh, PA, November 2004
6. K.Keahey, I.Foster, T.Freeman, X.Zhang, D.Garlon Wirtual Workspaces in the Grid, Europar 2005, Pisa, Italy, August, 2005.
7. I.Foster, J.Frey, S.Graham, S.Tuecke, K.Czajkowski, D.Ferguson, F.Leymann, M.Nally, I.Sedukhin, D.Snelling, T.Storey, W.Vambenepe, S.Weerawarana Modeling Stateful Resources with Web Services, version 1.1 http://www-128.ibm.com/developerworks/library/specification/ws-resource/, March 2004.
8. M.Jankowski, P.Wolniewicz, N.Meyer Virtual User System for Globus based grids, Cracow '04 Grid Workshop Proceedings, December 2004.
9. Ales Křenek and Zora Sebestianová. Perun – Fault-Tolerant Management of Grid Resources, Cracow '04 Grid Workshop Proceedings, December 2004.
10. Globus Toolkit Version 4: Software for Service-Oriented Systems. I. Foster. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005.

HLA Grid based support for simulation of vascular reconstruction

Katarzyna Rycerz[1], Marian Bubak[1,2], Maciej Malawski[1], Peter M.A. Sloot[3]

[1]Institute of Computer Science, AGH, al. Mickiewicza 30,30-059 Kraków, Poland
[2]Academic Computer Centre – CYFRONET, Nawojki 11,30-950 Kraków, Poland
[3]Faculty of Sciences, Section Computational Science, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{kzajac |bubak|malawski}@uci.agh.edu.pl, sloot@science.uva.nl
phone: (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54

Abstract. The Grid paradigm for distributed computation provides an interesting framework for the medical applications. We apply the High Level Architecture (HLA) model to the vascular reconstruction application, using distributed federations on the Grid for the communication among simulation and visualization components. HLA provides advanced features that allow to build collaborative environment where surgeons can exchange their knowledge and plan their operations. To achieve efficient execution of the Grid, we introduce a Grid HLA Management System (G-HLAM) that manages HLA–based simulations running on the Grid. This is done by introducing migration mechanisms for such applications.
Keywords HLA, Grid, distributed interactive simulations, federate management, medical simulation

1 Introduction

Distributed simulations often require extensive computing resources. The Grid [4, 5] is a promising means of solving this problem as it offers the possibility to use resources which are not centrally controlled and are under different administrative policies. Simulations built with an implementation of the High Level Architecture (HLA) system [6] allow for merging geographically-distributed parts (called *federates*) of simulations (called *federations*) into a coherent entity. The High Level Architecture is explicitly designed as support for interactive distributed simulations, it provides various services required for that specific purpose, such as time management, useful for time-driven or event-driven interactive simulations. It also takes care of data distribution management and enables all application components to see the entire application data space in an efficient way. On the other hand, the HLA standard does not provide automatic setup of HLA distributed applications and there is no mechanism for migrating federates according to the dynamic changes of host loads or failures, which is essential for Grid applications. Therefore, there is a need for a system that would manage HLA-based simulations on the Grid. The Grid Services concept provides a good

starting point for building the Grid HLA Management System (G-HLAM) for that purpose, as described in [10]. In this paper we show how the distributed vascular reconstruction simulation can benefit from the Grid by using G-HLAM. Such simulations remain a great challenge in medicine [12] and there is a need for a computer infrastructure that will significantly improve their performance. We believe that the Grid is a promising environment for such requirements, since it offers the possibility of accessing computational resources that have heretofore been inaccessible.

The paper is organized as follows: in Section 2 we present background of the addressed problem, in Section 3 we describe architecture and functionality of vascular reconstruction application, in Section 4 we describe benefits of using HLA for our purposes, in Section 5 we present G-HLAM system and describe how to use it in the application. In Section 6 experimental results are presented. We conclude in Section 7.

2 Background

Computer simulations in medicine are very important aid, especially in surgery [15]. This is because planning vascular operation is difficult task – the physician has to locate affected vessels, analyse them and then predict effect of the operation itself. Therefore, a verification of surgeon's decisions before the operation is very useful and allows for better patients' treatment [12]. These kind of simulations are very complex, require high performance execution and near real time interaction with its results during runtime [8].

Vascular disorders such as stenosis (narrowing of the artery by the accumulation of fat and cholesterol) and aneurysms (weakness of artery's wall which causes its ballooning) seriously influence the blood flow and can cause serious diseases [9]. Therefore, it is important to improve the flow quality in the affected vessels. Vascular reconstruction is a surgical procedure which redirects the blood flow from the affected place using a grafted bridge called a bypass.

Planning such operations requires that surgeon analyse the structure of artery, localize affected areas together with the optimal place to insert a bypass. Using computer simulations surgeons decisions can be verified before the actual operation takes place [7]. In this Section we present virtual reality based medical application [1] developed at Section Computational Science University of Amsterdam. We describe application modules and we outline three scenarios of its execution: a scenario when a single user runs only one simulation, extension for many simulations and finally, the collaborative environment.

3 Vascular reconstruction application

The input data for the application are obtained from various medical imaging techniques like X-ray angiography, computer tomography (CT) or MRI (magnetic resonance imaging). After getting digital images of patient vessels, the data is processed by four modules of the application as described below.

Analysis and segmentation module The goal of the segmentation process is to automatically find the lumen border between the blood and non-blood in input images [1]. Firstly, a wave front propagator algorithm is used to find an approximation of the centerline of the vessel. Next, the data are divided into a set of 2D slices orthogonal to the centerline. Then, in each slice a contour delineating the lumen border is detected. Finally, the stack of 2D contours is combined to 3D surface model, which is then passed to 3D editing tool.

3D editing tool This tool allows for editing stereoscopic images and executing experimental visualisation studies on realistic arteliar geometries [3]. A user can conduct measurements, pick up a position on the artery for creating a bypass and modify its shape and size. The final stage is generation of computational mesh. The prepared arteliar geometry, including aneurysms, bifurcations, bypasses and stents is converted into a coarse or fine (depending on the user) computational mesh that is then passed to the simulation module. The module is implemented using Visualisation Toolkit (Vtk) [13].

Simulation module Blood flow simulation is a parallel computational solver [2] that computes pressure, velocities, and shear stresses. The simulator is based on the Lattice–Boltzmann method (LBM) – a mesoscopic approach for simulating fluid flow based on the kinetic Boltzmann equation. The simulation produces intermediate results and sends them to visualisation module. The module is written is C and parallelised using MPI.

Visualisation and exploration module The visualization/exploration module uses a Virtual Reality Explorer (VRE) [3] , where the patient's data is visualized as a 3D stereoscopic image together with the graphical interpretation of the simulation results. The user can then manipulate the 3D images of arteries, patient's body and blood flow structures in virtual reality. VRE combines natural input modes of context sensitive interaction by voice, hand gestures and direct manipulation of virtual 3D objects. The interactive measurement component of the VRE provides the possibility to measure quantitatively distances, angles, diameters and some other parameters characterizing 3D objects in a virtual world, where markers are building blocks of distance, angle, and linestrip measurements. A desktop version of VRE which ports the basic functionality of VRE to the normal desktops is implemented using Vtk [13].

Workflow between application modules The input data for the application is got from X-ray angiography, computer tomography (CT) or MRI (magnetic resonance imaging). Then, a radiologist uses the module **Analysis and segmentation module** (module A) for analising raw digital images of vessels structures. At the first step, the information of affected vessels are segmented to obtain a 3D geometrical description of the arteries of interest. Next, the segmented artery is prepared for blood flow simulations in a 3D editing tool (module B), allowing to define in- and outlets, to filter and crop part of the artery, to add a by-pass, and to generate computational meshes as input to the blood flow simulators. Then, using module C – dedicated fluid flow solver – the time dependent blood flow in the artery is computed. The resulting flow, pressure and

shear stress fields are then shown to the user using a number of visualization techniques in a Virtual Environment (VE) - module D.

Need for a Grid The application modules require different resources: segmentation tool requires quick access to images' database, flow simulation requires computational power, editing and visualisation tools require specific VR hardware. It is quite unlikely to find those resources at one geographical site. Additionally, if more simulations need to be run concurrently, one site with computational power may be not sufficient. The similar problem arise, when many visualisations (users) located in different places want to observe the same simulation. Therefore, the application modules usually have to be located in geographically different places and the Grid concept that facilitates access to computing resources may be a very promising approach here.

Collaborative environment for vascular reconstruction The collaborative environment is needed when group of surgeons from different hospitals want to discuss together interesting medical cases and exchange ideas about planning difficult operation. In Fig. 1 we show the situation for two users, but it can be easily extended for more. Each of users starts number of simulations as in previous case and becomes their owner - that means he can fully control them as in previous case. However, the user can also interact with not owned simulations in two ways. Firstly, by requesting the output data of not owned simulation – in this case, there is no problem with concurrent data access, since still only one user can change the data. Secondly, by requesting to control not owned simulation - the concurrent data access has to be controlled by communication bus. The user can: stop/pause the chosen simulation during runtime, switch between different simulations owned by him (receiving output), switch ownership of simulations (ability to stop/pause particular simulation), restart

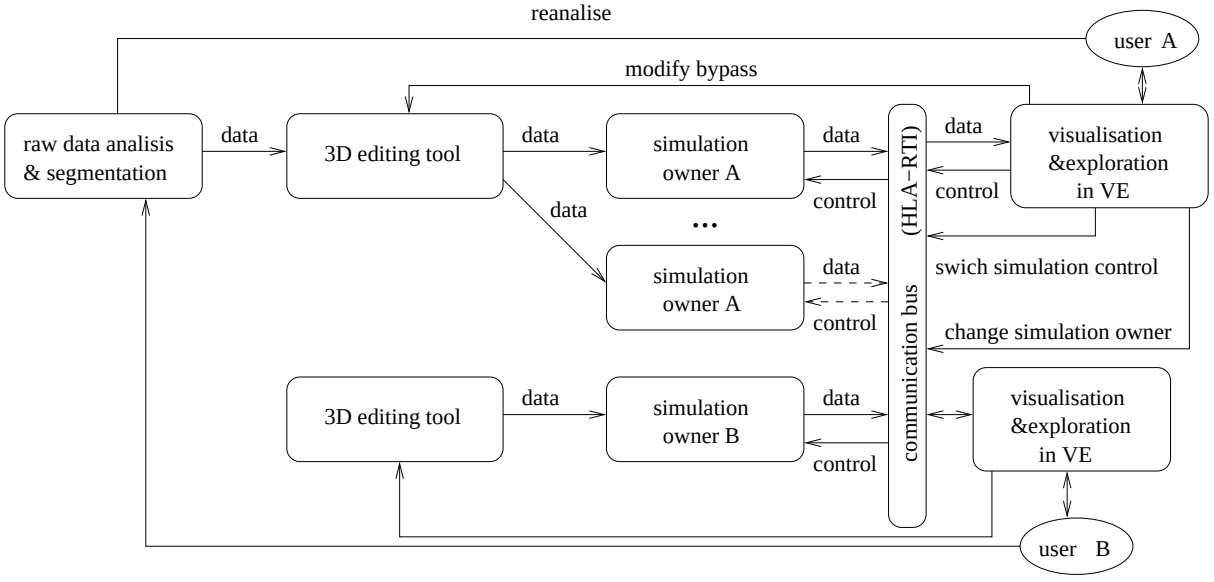


Fig. 1. Scenario with multiple user and multiple simulation

with different bypass and reanalyse raw data and remake segmentation.

4 HLA–based communication

HLA is very convenient solution for that connection of simulation modules with visualisation modules as it allows for : **building geographically distributed system** - as stated in Section 3, usually there is a need to place simulation in different place then visualisation, especially if there are more than one as in the Fig. 1, **dynamic joining of simulation/visualisation to the application** – useful when a user A in the Fig. 1 needs to start simulations one by one, or the user B joins lately and starts his own simulations, but wants to share results with others, **easy switching** between background (indicated with dashed arrows in the Fig. 1) and foreground (indicated with solid arrows) simulations - useful, when the user decides to get output data from different simulation; using HLA he can unsubscribe from data currently received and subscribe to data of the new simulation, **dynamic resigning of simulation/visualisation from application** – the user can quit the application or stop some of simulations without disturbing others, **notification** - useful for notifying the simulation about the user commands i.e. pause, cancellation, **easy switching between ownership** - useful when a user that created a simulation wants to give its control away to a different user, **concurrency control** – related to ownership – assures that only one user can control the simulation at the time.

According to scenarios presented above and types of interaction within the application, these HLA features are very useful and fill application requirement to communication infrastructure between simulations and visualisations.

5 The Grid-based HLA Management System

5.1 Overview of the system

The Grid HLA Management System (G-HLAM) supports efficient execution of HLA-based applications on top of the Open Grid Services Infrastructure as presented in [14, 10, 11].

The group of main G-HLAM services consists of: a *Broker Service* which coordinates management of the simulation, a *Performance Decision Service* which decides when the performance of any of the federates is not satisfactory and therefore migration is required, and a *Registry Service* which stores information about the location of local services. On each Grid site, supporting HLA, there are local services for performing migration commands on behalf of the *Broker Service*, as well as for monitoring federates and benchmarking. The *HLA–Speaking Service* is one of the local services interfacing federates with the G-HLAM system, using GRAM for submission of federates. A more detailed description of the *HLA-Speaking Service*, together with the GridHLAController library, which actually interfaces the application code with the system, can be found in [11].

5.2 Application within G-HLAM

For purposes of this case study, we have chosen two modules of described application: simulation and visualisation/exploration. G-HLAM usage is depicted in Fig. 2. As described above, and shown in the figure, the case study application

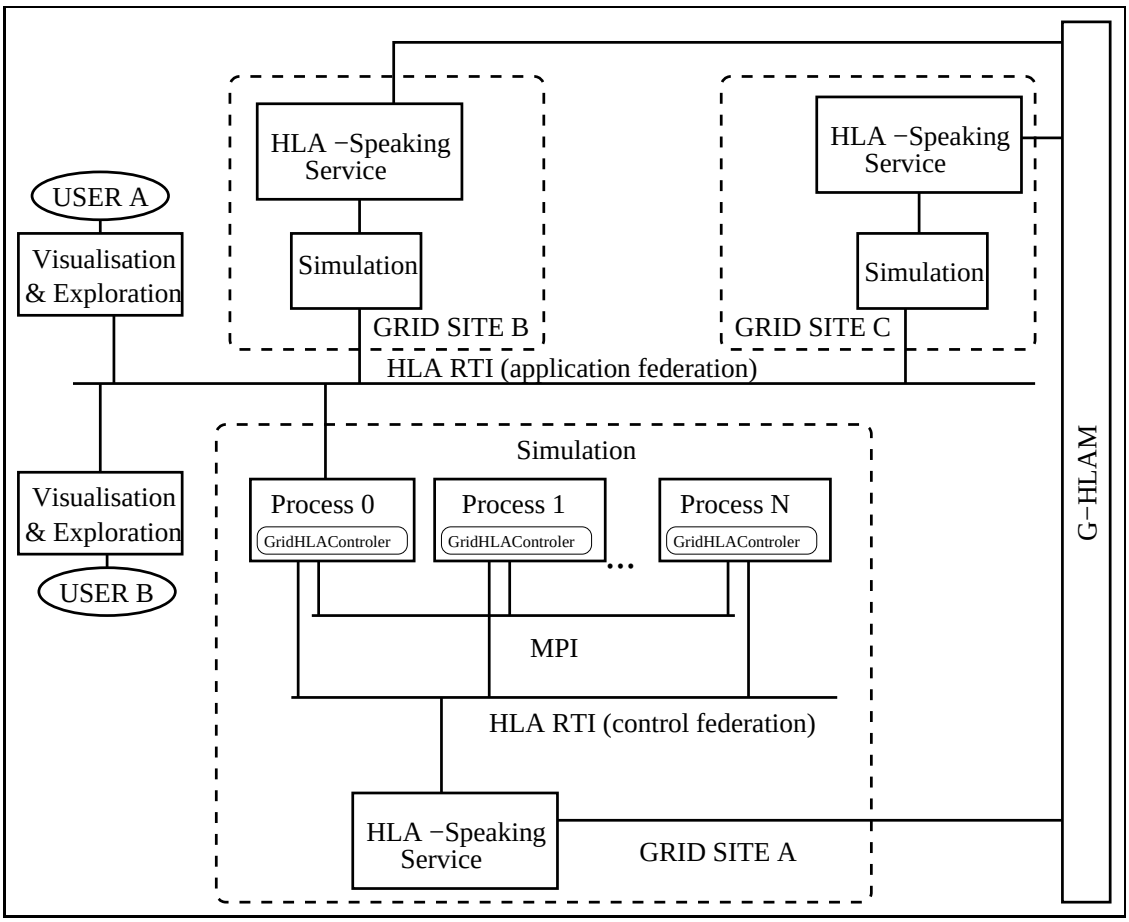


Fig. 2. G-HLAM for the medical application

consists of two kinds of modules: an MPI Lattice Boltzman simulation module and an integrated visualization–exploration module. For simplicity, in Fig.2 we have presented an example with three simulations and two visualisations, but it can be easily extended. Moreover, we outline details of simulation only on Grid site A, however the same structure applies to site B and C, where simulation is shown as a one box.

As can be seen, there are two kinds of HLA federations : one application federation and N internal G-HLAM control federations, where N is a number of Grid sites used by simulation modules.

Application federation. Communication between application components is performed by HLA, so in the Fig.2 all simulations and visualisations are connected to application federation. Simulations are connected to application federation only by their MPI root processes that are responsible for sending data

to visualisation. This federation is specific to the application and is designed by its developer.

G-HLAM Control Federations. On each site, there is a *HLA–Speaking Service* for controlling simulation by G-HLAM The service is an interface between G-HLAM and application federates. It starts the application processes on its site and sends them saving/restoring request from G-HLAM when there is a need for migration. Each site has its own control federation for communicating with the *HLA–Speaking service* residing on this site and all simulation processes join respective control federations by GridHLAController library. The library is an interface between user code and G-HLAM, it passes saving and restoring request to the user code and assures that all user processes behave correctly, when one of them is migrated.

6 Results

In this Section we present results from two experiments using G-HLAM for the prototype vascular reconstruction application. The prototype consisted of two types of modules communicating with HLA: *simulation modules* (MPI parallel simulation) and *visualization–receiver modules* (responsible for receiving data from simulation). As above, we use the last scenario of the application (collaborative environment) since other scenarios are just specific cases thereof. First, we show how migration time scales along with the number of simulation and visualization–receiver modules in the collaborative environment. Next, we show how migration improves performance from the point of view of the user - i.e. how sending output data from the simulation changes after migration if the partial simulation results are actually watched by someone. The experiments were performed on the DutchGrid DAS2 testbed infrastructure and at CYFRONET, Krakow, as shown in Tab. 1.

Operating System	Red Hat Enterprise Linux Advanced Server, version 3		
Network	10 Gbps (DAS2) + 155 Mbps (DAS2-Cyfronet)		
Role	**Name**	**CPU**	**RAM**
Migration source	DAS2 Nikhef	Pentium III 1 GHz	1 GB
Migration destination	DAS2 Leiden	Pentium III 1 GHz	2 GB
other visualizations and simulations	DAS2 Delft	Pentium III 2GHz	2 GB
	DAS2 Utrecht	Pentium III 1 GHz	1 GB
	DAS2 Vrije	Pentium III 1 GHz	2 GB
RTIexec	Cyf Krakow	Xeon 2.4 GHz	1 GB

Table 1. Grid testbed infrastructure

In the first experiment we ran four simulations (each containing 12 MPI processes)in our collaborative environment and the number of visualization–receivers varied from 3 to 22. We assigned visualization–receivers for each simulation in the most balanced way possible, so that each simulation had an equal

number of visualisation-receivers collecting its data (exact to the remainder of dividing the number of visualizations and the number of simulations). We then migrated one of the simulations.

In the second experiment one simulation was migrated. The number of visualization–receivers was fixed and equal 25.

We observed that the type of module (simulation or visualization–receiver) does not have any impact on migration time. This is because only one (master) process of the parallel simulation participates in the application federation and its role is equal to a single visualisation–receiver process. Therefore, we plot migration time as a function of all federates in the application federation regardless of their type.

In our experiments, in each step, the simulation produces 52000 velocity vectors of simulated blood flow in 3D space. For our experiments we used GT v3.2 and HLA RTI 1.3v5. The results were obtained as an average from 10 measurements. The error bars indicate estimated standard deviation.

Migration Overhead From our results it can be seen that migration overhead is linear with respect to the number of federates in the application federation (modules in the collaborative environment) when using reliable transport in the HLA RTI implementation. Basing on our other experiments performed with migration of N–body simulation [11] we can say that this is probably because the most time consuming operation is the rejoining application federation. The federate has to open TCP connections to all other federates in the federation. In our approximating linear function $A = 2.7$, $\sigma_A = 0.8$ and $B = 87.0$, $\sigma_B = 13$. To check if the approximation is appropriate, we performed a χ^2 test for 11 degrees of freedom. For our data $\chi^2 = 8.7$, which is lower than the critical value 17.2 for significance level 0.1.

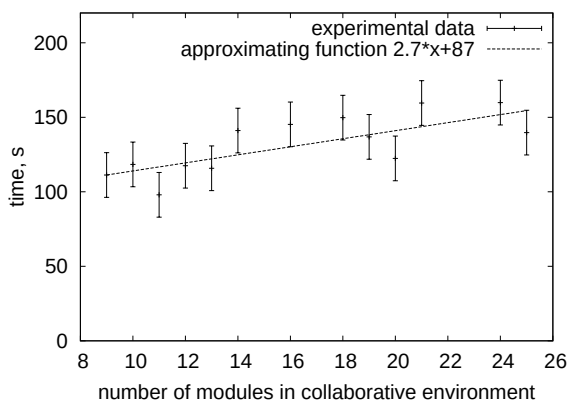


Fig. 3. Total migration time as a function of the number of modules in collaborative environment federates

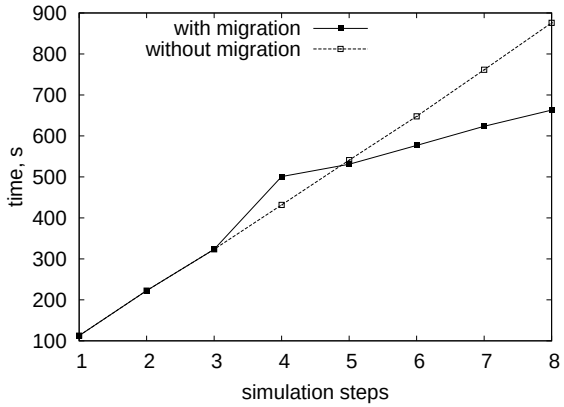


Fig. 4. Impact of migration on simulation performance within collaborative environment

Impact of migration on performance of simulation within the collaborative environment In this experiment we show how migration can improve the efficiency of simulation execution when its results are sent on–line to many

users. the bandwidth available for testing was broad (10Gbps), so communication did not play an important role and calculations were the most time–consuming part of the execution. In order to create conditions in which migration would be useful, we increased the load of the Grid site where the simulation was executed (cluster in Amsterdam) by submitting non–related, computationally–intensive jobs. Next, we imitated a Resource Broker and migrated the simulation to another site which was not overloaded (cluster in Leiden). The experiments were performed at night in order to avoid interference with other users and repeated 10 times to check if they were reproducible. Fig. 4 shows the time as a function of the number of interactive steps with a human in the loop (for the first 8 steps). At each step, the simulation calculates data and sends it to the 25 visualization–receivers modules using HLA. The dashed line shows the execution time of the simulation steps in the case when the simulation was not migrated. In this situation it is better to spend some time on migration to another site, from where the response time is shorter, as shown by the solid line in the figure. Fig. 4 shows that the human can gain access time between steps 4 and 5, independently of the time lost on migration (performed between steps 3 and 4).

7 Summary and Conclusions

This paper presented how collaborative environment for supporting planning vascular reconstruction operations can benefit from both HLA standard and the Grid environment. In the paper we have shown that HLA provides mechanisms to build advanced collaborative environments. The most important features of HLA include synchronisation mechanisms, data distribution management and ownership management. Also, HLA allows for building geographically distributed simulation systems in a relatively easy way.

However, vascular reconstruction application needs also to take advantage from the Grid and distributed resources provided by it. Therefore, we show how G-HLAM system can be used to manage efficient execution of HLA–based application on the Grid environment. This is done by migration of badly performing parts of simulations to a better location in order to reduce computation and communication time and effectively improving the overall performance.

Acknowledgments The authors would like to thank Piotr Nowakowski for useful remarks. This research is partly funded by the EU IST Project CoreGRID, the Polish State Committee for Scientific Research SPUB-M grant, and by the Dutch Virtual Laboratory for e-Science project (www.vl-e.nl).

References

1. L. Abrahamyan, J.A. Schaap, A.G. Hoekstra, D.P. Shamonin, F.M.A. Box, R.J. van der Geest, J.H.C. Reiber, and P.M.A. Sloot. A Problem Solving Environment for Image-Based Computational Hemodynamics. In V.S. Sunderam, G.D. van Albada, P.M.A. Sloot, and J.J. Dongarra, editors, *Computational Science - ICCS*

2005: 5th International Conference, Atlanta, GA, USA, Proceedings, Part I, volume 3514 of *Lecture Notes in Computer Science*, pages 287–294, Berlin, Heidelberg, May 2005. Springer.

2. A.G. Artoli, A.M. Hoekstra and P.M.A. Sloot. Simulation of a systolic cycle in a realistic artery with the Lattice Boltzmann BGK method. *Int. J. Mod. Phys. B*, 17:95–98, 2003.
3. R.G. Belleman. *Interactive Exploration in Virtual Environments.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, April 2003. Promotor: Prof. Dr. P.M.A. Sloot.
4. I. Foster. What is the Grid? A three checkpoints list. *GridToday Daily News And Information For The Global Grid Community*, 1(6), July 2002.
5. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002. `http://www.globus.org/research/papers.html`.
6. HLA specification. `http://www.sisostds.org/stdsdev/hla/`.
7. Kevin Montgomery, Michael Stephanides, Stephen Schendel, and Muriel Ross. A case study using the virtual environment for reconstructive surgery. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 431–434, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.
8. Steven Pieper, Joseph Rosen, and David Zeltzer. Interactive graphics for plastic surgery: a task-level analysis and implementation. In *SI3D '92: Proceedings of the 1992 symposium on Interactive 3D graphics*, pages 127–134, New York, NY, USA, 1992. ACM Press.
9. P.M. Rothwell, J. Slattery, and C.P. Warlow. A Systematic Comparison of the Risks of Stroke and Death Due to Carotid Endarterectomy for Symptomatic and Asymptomatic Stenosis. *Stroke*, 27(2):266–269, 1996.
10. K. Rycerz, M. Bubak, M. Malawski, and P.M.A. Sloot. A Framework for HLA-Based Interactive Simulations on the Grid. *SIMULATION*, 81(1):67–76, 2005.
11. K. Rycerz, M. Bubak, M. Malawski and P. Sloot. A Grid Service for Management of Multiple HLA Federate Processes. submitted to PPAM conference, Poznan, 2005.
12. P.M.A. Sloot, A. Tirado-Ramos, A.G. Hoekstra, and M. Bubak. Interactive Grid Environment for Non-Invasive Vascular Reconstruction. In *2nd International Workshop on Biomedical Computations on the Grid (BioGrid'04), in conjunction with Fourth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2004)*. IEEE, April 2004.
13. Visualisation toolkit home page. http://public.kitware.com/VTK/.
14. K. Zając, M. Bubak, M. Malawski, and P.M.A Sloot. Towards a Grid Management System for HLA-Based Interactive Simulations. In S.J. Turner and S.J.E. Taylor, editor, *Proceedings Seventh IEEE International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2003)*, pages 4–11, Delft, The Netherlands, October 2003. IEEE Computer Society.
15. Z. Zhao. *An agent based architecture for constructing Interactive Simulation Systems.* PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, December 2004. Promotor: Prof. Dr. P.M.A. Sloot, Co-promotor: Dr. G.D. van Albada.

Fault-Injection and Dependability Benchmarking for Grid Computing Middleware

Sébastien Tixeuil, [1], Luis Moura Silva[2],

William Hoarau[1], Gonçalo Jesus[2], João Bento[2], Frederico Telles[2]

[1] LRI – CNRS UMR 8623 & INRIA Grand Large,
Université Paris Sud XI, France
Email : tixeuil@lri.fr

[2] Departamento Engenharia Informática, Universidade de Coimbra,
Polo II, 3030-Coimbra, Portugal
Email: luis@dei.uc.pt

Abstract. In this paper we will present some work on dependability benchmarking for Grid Computing that represents a common view between two groups of Core-Grid: INRIA-Grand Large and University of Coimbra. We present a brief overview of the state of the art, followed by a presentation of the FAIL-FCI system from INRIA that provides a tool for fault-injection in large distributed systems. Then we present DBGS, a dependable Benchmark for Grid Services. We conclude the paper with some considerations about the avenues of research ahead that both groups would like to contribute, on behalf of the Core-GRID network.

1 Introduction

One of the topics of paramount importance in the development of Grid middleware is the impact of faults since their probability of occurrence in a Grid infrastructure and in large-scale distributed system is actually very high. So it is mandatory that Grid middleware should be itself reliable and should provide a comprehensive support for fault-tolerance mechanisms, like failure-detection, checkpointing, replication, software rejuvenation, component-based reconfiguration, among others. One of the techniques to evaluate the effectiveness of those fault-tolerance mechanisms and the reliability level of the Grid middleware it to make use of some fault-injection tool and robustness tester to conduct some experimental assessment of the dependability metrics of the target system. In this paper, we will present a software fault-injection tool and a workload generator for Grid Services that can be used for dependability benchmarking in Grid Computing.

The final goal of our common work is to provide some contributions for the definition of a dependability-benchmark for Grid computing and to provide a set of

tools and techniques that can be used by the developers of Grid middleware and Grid-based applications to conduct some dependability benchmarking of their systems.

In this paper we present a fault-injection tool for large-scale distributed systems (developed developed by INRIA-GrandLarge) and a workload generator for Grid Services (being developed by the University of Coimbra) that include those four components mentioned before. To the best of our knowledge the combination of these two tools represent the most complete testbed for dependability benchmarking of Grid applications.

The remainder of this paper is organized as follows. Section 2 describes a summary of the related work. Section 3 describes the FAIL-FCI infrastructure from INRIA. Section 4 briefly describes DBGS, a dependability benchmarking tool for Grid Services. Section 5 concludes the paper.

2 Related Work

In this section we present a summary of the state-of-the-art in the two main topics of this paper: dependability benchmarking and fault-injection tools.

2.1 Dependability Benchmarking

The idea of dependability benchmarking is now a hot-topic of research [1] and there are already several publications in the literature. In [2] it is proposed a dependability benchmark for transactional systems (DBench-OLTP). Another dependability benchmark for transactional systems is proposed in [3]. This one considered a faultload based on hardware faults. A dependability benchmark for operating systems is proposed by [4]. Research work developed at Berkeley University has lead to the proposal of a dependability benchmark to assess human-assisted recovery processes [5]. The work carried out in the context of the Special Interest Group on Dependability Benchmarking (SIGDeB), created by the IFIP WG 10.4, has resulted in a set of standardized availability classes to benchmark database and transactional servers [6]. Research work at Sun Microsystems defined a high-level framework [7] dedicated specifically to availability benchmarking. Within this framework, they have developed two benchmarks: one benchmark [8] that addresses specific aspects of a system's robustness on handling maintenance events such as the replacement of a failed hardware component or the installation of software patch; and another benchmark is related to system recovery [9]. At IBM, the Autonomic Computing initiative is also developing benchmarks to quantify a system's level of autonomic capability, addressing four main spaces of IBM's self-management: self-configuration, self-healing, self-optimization, and self-protection [10]. We are looking with detail into this initiative and our aim will be to introduce some of these metrics in Grid middleware to reduce the maintenance burden and to increase the availability of Grid applications in production environments. Finally, in [11] the authors present a

dependability benchmark for Web-Servers. In some way we follow this trend by developing a benchmark for SOAP-based Grid services.

2.2 Fault-injection Tools

When considering solutions for software fault injection in distributed systems, there are several important parameters to consider. The main criterion is the usability of the fault injection platform. If it is more difficult to write fault scenarios than to actually write the tested applications, those fault scenarios are likely to be dropped from the set of performed tests. The issues in testing component-based distributed systems have already been described and methodology for testing components and systems has already been proposed [12-13]. However, testing for fault tolerance remains a challenging issue. Indeed, in available systems, the fault-recovery code is rarely executed in the test-bed as faults rarely get triggered. As the ability of a system to perform well in the presence of faults depends on the correctness of the fault-recovery code, it is mandatory to actually test this code. Testing based on fault-injection can be used to test for fault-tolerance by injecting faults into a system under test and observing its behavior. The most obvious point is that simple tests (*e.g.* every few minutes or so, a randomly chosen machine crashes) should be simple to write and deploy. On the other hand, it should be possible to inject faults for very specific cases (*e.g.* in a particular global state of the application), even if it requires a better understanding of the tested application. Also, decoupling the fault injection platform from the tested application is a desirable property, as different groups can concentrate on different aspects of fault-tolerance.

Decoupling requires that no source code modification of the tested application should be necessary to inject faults. Also, having experts in fault-tolerance test particular scenarios for application they have no knowledge of favors describing fault scenarios using a high-level language, that abstract practical issues such that communications and scheduling. Finally, to properly evaluate a distributed application in the context of faults, the impact of the fault injection platform should be kept low, even if the number of machines is high. Of course, the impact is doomed to increase with the complexity of the fault scenario, *e.g.* when every action of every processor is likely to trigger a fault action, injecting those faults will induce an overhead that is certainly not negligible.

Several fault injectors for distributed systems already exist. Some of them are dedicated to distributed real-time systems such as DOCTOR [14]. ORCHESTRA [15] is a fault injection tool that allows the user to test the reliability and the liveliness of distributed protocols. ORCHESTRA is a "*Message-level fault injector*" because a fault injection layer is inserted between two layers in the protocol stack. This kind of fault injector allows injecting faults without requiring the modification of the protocol source code. However, the expressiveness of the faults scenario is limited because there is no communication between the various state machines executed on every node. Then, as the faults injection is based on exchanged messages, the knowledge of the type and the size of these messages is required. Nevertheless, those approaches do not fit the cluster and Grid category of applications.

The NFTAPE project [16] arose from the double observation that no tool is sufficient to inject all fault models and that it is difficult to port a particular tool to different systems. Although NFTAPE is modular and very portable, the choice of a completely centralized decision process makes it very intrusive (its execution strongly perturbs the system being tested). Finally, writing a scenario quickly becomes complex because of the centralized nature of the decisions during the tests when they imply numerous nodes.

LOKI [17] is a fault injector dedicated to distributed systems. It is based on a partial view of the global state of the distributed system. An analysis *a posteriori* is executed at the end of the test to infer a global schedule from the various partial views and then verify if faults were correctly injected (*i.e.* according to the planned scenario). However, LOKI requires the modification of the source code of the tested application. Furthermore, faults scenario are only based on the global state of the system and it is difficult (if not impossible) to specify more complex faults scenario (for example injecting "cascading" faults). Also, in LOKI there is no support for randomized fault injection.

In [18] is presented Mendosus, a fault-injection tool for system-area networks that is based on the emulation of clusters of computers and different network configurations. This tool made some first steps in the fault-injection and assessment of faults in large distributed systems, although FCI has made some steps ahead.

Finally in [19] is presented a fault-injection tool that was specially developed to assess the dependability of Grid (OGSA) middleware. This is the work more related with ours and we welcome the first contributions done by those authors in the area of grid middleware dependability. However, the tool described in that paper is very limited since it only allows the injection of faults in the XML messages in the OGSA middleware, which seems to be a bit far from the real faults experienced in real systems.

In the rest of the paper we will present two tools for fault-injection and workload generation that complement each other quite well, and if used together might represent an interesting package to be used by developers of Grid middleware and applications.

3 FAIL-FCI Framework from INRIA

In this section, we describe the FAIL-FCI framework from INRIA. First, FAIL (for FAult Injection Language) is a language that permits to easily described fault scenarios. Second, FCI (for FAIL Cluster Implementation) is a distributed fault injection platform whose input language for describing fault scenarios is FAIL. Both components are developed as part of the Grid eXplorer project [20] which aims at emulating large-scale networks on smaller clusters or grids.

The FAIL language allows defining fault scenarios. A scenario describes, using a high-level abstract language, state machines which model fault occurrences. The FAIL language also describes the association between these state machines and a computer (or a group of computers) in the network. The FCI platform (see **Figure 1**) is composed of several building blocks:

1. **The FCI compiler**: The fault scenarios written in FAIL are pre-compiled by the FCI compiler which generates C++ source files and default configuration files.
2. **The FCI library**: The files generated by the FCI compiler are bundled with the FCI library into several archives, and then distributed across the network to the target machines according to the user-defined configuration files. Both the FCI compiler generated files and the FCI library files are provided as source code archives, to enable support for heterogeneous clusters.
3. **The FCI daemon**: The source files that have been distributed to the target machines are then extracted and compiled to generate specific executable files for every computer in the system. Those executables are referred to as the FCI daemons. When the experiment begins, the distributed application to be tested is executed through the FCI daemon installed on every computer, to allow its instrumentation and its handling according to the fault scenario.

Our approach is based on the use of a software debugger. Like the Mantis parallel debugger [21], FCI communicates to and from `gdb` (the Free Software Foundation's portable sequential debugging environment) through Unix pipes. But contrary to Mantis approach, communications with the debugger must be kept to a minimum to guarantee low overhead of the fault injection platform (in our approach, the debugger is only used to trigger and inject software faults). The tested application can be interrupted when it calls a particular function or upon executing a particular line of its source code. Its execution can be resumed depending on the considered fault scenario.

With FCI, every physical machine is associated to a fault injection daemon. The fault scenario is described in a high-level language and compiled to obtain a C++ code which will be distributed on the machines participating to the experiment. This C++ code is compiled on every machine to generate the fault injection daemon. Once this preliminary task has been performed, the experience is then ready to be launched. The daemon associated to a particular computer consists in:

1. a state machine implementing the fault scenario,
2. a module for communicating with the other daemons (*e.g.* to inject faults based on a global state of the system),
3. a module for time-management (*e.g.* to allow time-based fault injection),
4. a module to instrument the tested application (by driving the debugger), and
5. a module for managing events (to trigger faults).

FCI is thus a Debugger-based Fault Injector because the injection of faults and the instrumentation of the tested application is made using a debugger. This makes it possible not to have to modify the source code of the tested application, while enabling the possibility of injecting arbitrary faults (modification of the program counter or the local variables to simulate a buffer overflow attack, etc.). From the user point of view, it is sufficient to specify a fault scenario written in FAIL to define an experiment. The source code of the fault injection daemons is automatically generated. These daemons communicate between them explicitly according to the

user-defined scenario. This allows the injection of faults based either on a global state of the system or on more complex mechanisms involving several machines (*e.g.* a cascading fault injection). In addition, the fully distributed architecture of the FCI daemons makes it scalable, which is necessary in the context of emulating large-scale distributed systems.

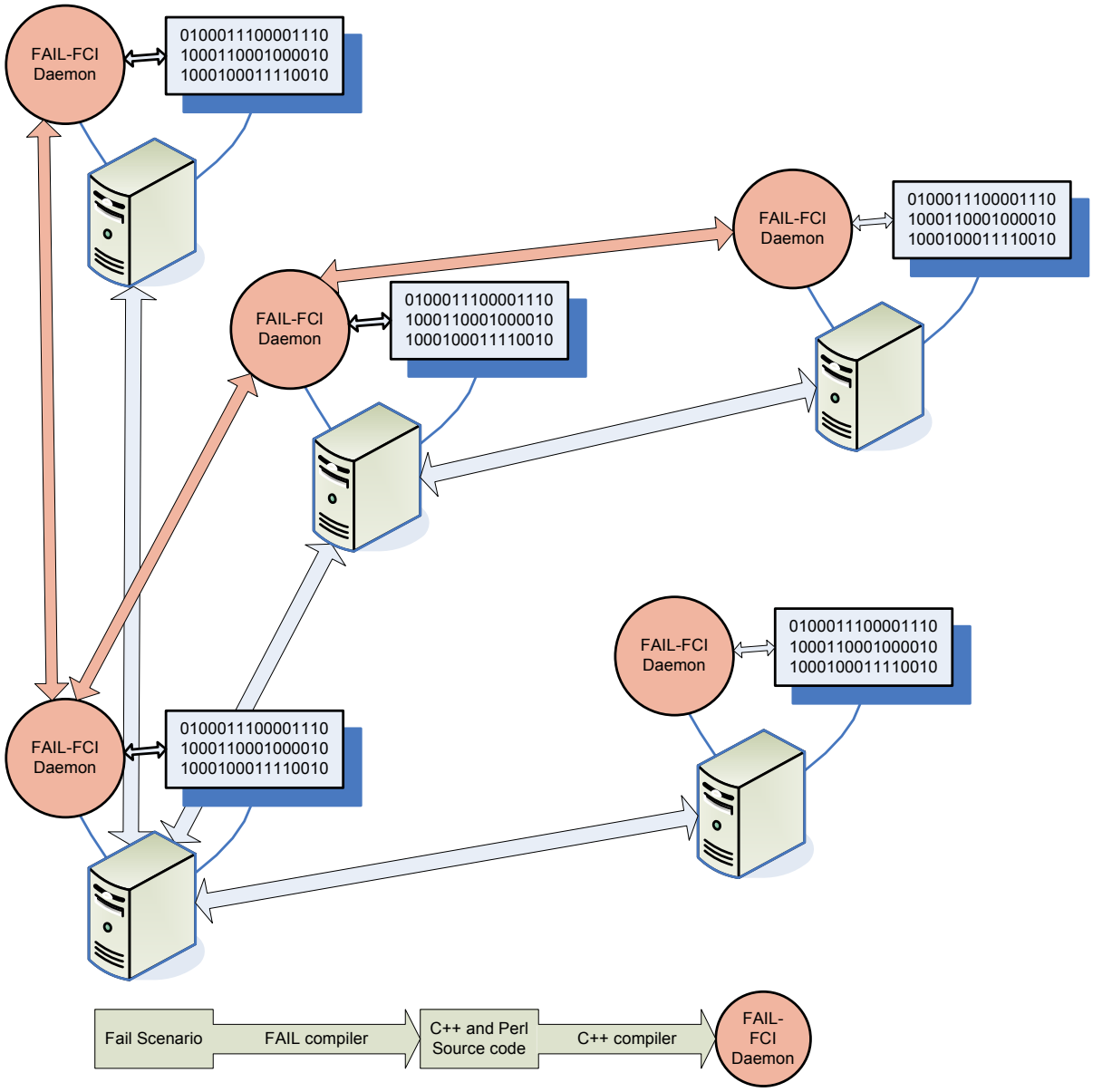


Figure 1: the FCI Platform

FCI daemons have two operating modes: a random mode and a deterministic mode. These two modes allow fault injection based on a probabilistic fault scenario (for the first case) or based on a deterministic and reproducible fault scenario (for the second case). Using a debugger to trigger faults also permits to limit the intrusion of the fault injector during the experiment. Indeed, the debugger places breakpoints which correspond to the user-defined fault scenario and then runs the tested application. As

long as no breakpoint is reached the application runs normally and the debugger remains inactive.

Fail-FCI has been used to assess the dependability of XtremWeb [22] and some results are being collected that allow us to assess the effectiveness of some fault-tolerance techniques that can be applied to desktop grids.

4 DBGS: Dependability Benchmark for Grid Services

DBGS is a dependability benchmark for Grid Services that follow the OGSA specification [23]. Since the OGSA model is based on SOAP technology we have developed a benchmark tool for SOAP-based services. This benchmark includes the four components, mentioned in section 1: (a) definition of a workload to the system under test (SUT); (b) optional definition of a faultload to the SUT system; (c) collection and definition of the benchmark measurements; (d) definition of the benchmark procedures. The DGGS is composed by the following components presented in Figure 2.

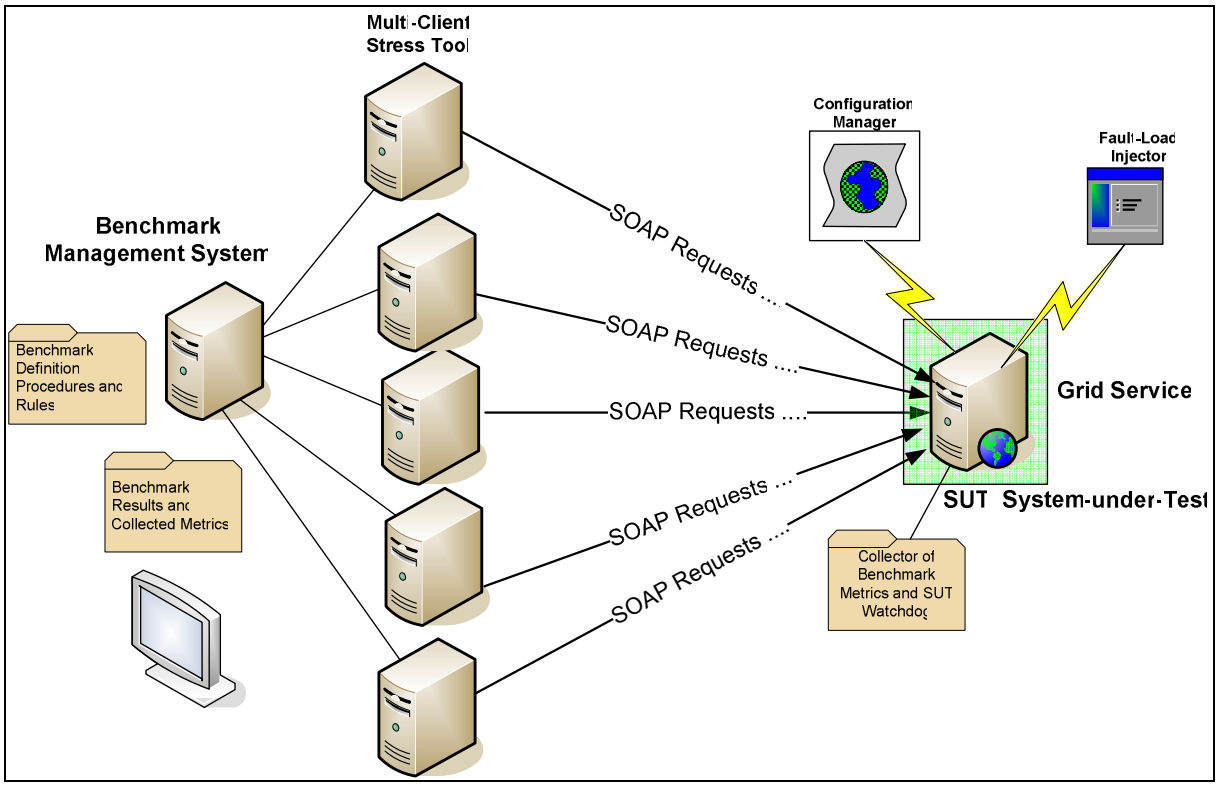


Figure 2: Experimental setup overview of the DBGS benchmark.

The system-under-test (SUT) consists of a SOAP server running some Grid or Web-Service. From the point of view of the benchmark the SUT corresponds to an application server, a SOAP router and a Grid service that will execute under some workload, and optionally will be affected by some fault-load.

The Benchmark Management System (BMS) is a collection of software tools that allows the automatic execution of the benchmark. It includes a module for the definition of the benchmark, a set of procedures and rules, definition of the workload

that will be produced in the SUT, a module that collects all the benchmark results and produces some results that are expressed as a set of dependability metrics. The BMS system may activate a set of clients (running in separate machines) that inject the defined workload in the SUT by making SOAP requests to the end Grid Service. All the execution of these client machines is timely synchronized and all the partial results collected by each individual client are merged into a global set of results that generated the final assessment of the dependability metrics. The BMS system includes a reporting tool that presents the final results in a readable and graphic format.

The results generated by each benchmark run are expressed as throughput-over-time (requests-per-second in a time axis), the total turnaround time of the execution, the average latency, the functionality of the services, the occurrence of failures in the Grid service/server, the characterization of those failures (crash, hang, zombie-server), the correctness of the final results at the server side and the failure scenarios that are observed at the client machines (explicit SOAP error messages or time-outs).

From the side of the SUT system, there are four modules that also make part of the DBGS benchmark: a fault-load injector, a configuration manager, a collector of benchmark results and a watchdog of the SUT system.

The fault-load injector does not inject faults directly in the software like the fault-injection tools, previously mentioned in section 2. This injector only produces some impact at the operating system level: it consumes resources from the operating system like memory, threads, file-handles, database-connections, sockets. We have observed that Grid and WS middleware is not robust enough because the underlying middleware (e.g. Application server and the SOAP implementation) is very unreliable when there are lack of operating system resources, like memory leakage, memory exhaustion and over-creation of threads. These are the scenarios we want to generate with this fault-load module. This means that software bugs are not directly emulated by this module, but rather by a tool like FAIL-FCI.

The configuration manager helps in the definition of the configuration parameters of the SUT middleware. It is absolutely that the configuration parameters may have a considerable impact in the robustness of the SUT system. By changing those parameters in different runs of the benchmark it allow us to assess the impact of those parameters in the results expressed as dependability metrics.

Finally, the SUT system should also be installed with a module to collect raw data from the benchmark execution. This data will be then sent to the BMS server that will merge and compare with the data collected from the client machines. The final module is a SUT-Watchdog that detects when a SUT system crashes or hangs when the benchmark is executing. When a crash or hang is detected the watchdog generates a restart of the SUT system and associated applications, thereby allowing an automatic execution of the benchmark runs without user intervention.

We have been collecting a large set of experimental results with this tool. The results are not presented here for lack of space, but in summary, we can say that this benchmark tool allowed us to spot some of the software leaks that can be found in current implementations of SOAP that are currently being used in Grid services and those problems may completely undermine the dependability level of the Grid applications.

5 Conclusions and Future Work

This paper presented a fault-injection tool for large-scale distributed systems that is currently being used to measure the fault-tolerance capabilities included in XtremWeb, and a second tool that can be directly used for dependability benchmarking of Grid Services that follow the OGSA model, and are thereby implemented by using SOAP technology. These two tools together fit quite well, since their target is really complementary. We feel that these two groups of Core-GRID will provide some valuable contribution in the area of dependability benchmarking for Grid Computing, and our work in cooperation has a long avenue ahead with several research challenges. At the end of the road we hope to have contributed to increase the dependability of Grid middleware and applications by the deployment of these tools to the community.

6 Acknowledgements

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

References

1. P.Koopman, H.Madeira. “Dependability Benchmarking & Prediction: A Grand Challenge Technology Problem”, Proc. 1st IEEE Int. Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems; Phoenix, Arizona, USA, Nov 1999
2. M. Vieira and H. Madeira, “A Dependability Benchmark for OLTP Application Environments”, Proc. 29th Int. Conf. on Very Large Data Bases (VLDB-03), Berlin, Germany, 2003.
3. K. Buchacker and O. Tschaeche, “TPC Benchmark-c version 5.2 Dependability Benchmark Extensions”, http://www.faumachine.org/papers/tpcc-depend.pdf, 2003.
4. A. Kalakech, K. Kanoun, Y. Crouzet and A. Arlat. “Benchmarking the Dependability of Windows NT, 2000 and XP”, Proc. Int. Conf. on Dependable Systems and Networks (DSN 2004), Florence, Italy, IEEE CS Press, 2004.
5. A. Brown, L. Chung, W. Kakes, C. Ling, D. A. Patterson, "Dependability Benchmarking of Human-Assisted Recovery Processes", Dependable Computing and Communications, DSN 2004, Florence, Italy, June, 2004

6. D. Wilson, B. Murphy and L. Spainhower. “Progress on Deining Standardized Classes of Computing the Dependability of Computer Systems”, Proc. DSN 2002, Workshop on Dependability Benchmarking, Washington, D.C., USA, 2002.
7. J. Zhu, J. Mauro, I. Pramanick. “R3 - A Framwork for Availability Benchmarking,” Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), USA, 2003.
8. Ji J. Zhu, J. Mauro, and I. Pramanick, “Robustness Benchmarking for Hardware Maintenance Events”, in Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003), pp. 115-122, San Francisco, CA, USA, IEEE CS Press, 2003.
9. J. Mauro, J. Zhu, I. Pramanick. “The System Recovery Benchmark,” in Proc. 2004 Pacific Rim Int. Symp. on Dependable Computing, Papeete, Polynesia, 2004.
10. S. Lightstone, J. Hellerstein, W. Tetzlaff, P. Janson, E. Lassettre, C. Norton, B. Rajaraman and L. Spainhower. "Towards Benchmarking Autonomic Computing Maturity", 1st IEEE Conf. on Industrial Automatics (INDIN-2003), Canada, August 2003.
11. J. Durães, M. Vieira and H. Madeira. "Dependability Benchmarking of Web-Servers", Proc. 23rd International Conference, SAFECOMP 2004, Potsdam, Germany, September 2004. Lecture Notes in Computer Science, Volume 3219/2004
12. S Ghosh, AP Mathur, "Issues in Testing Distributed Component-Based Systems", 1st Int. ICSE Workshop on Testing Distributed Component-Based Systems, 1999
13. H. Madeira, M. Zenha Rela, F. Moreira, and J. G. Silva. “Rifle: A general purpose pin-level fault injector”. In European Dependable Computing Conference, pages 199–216, 1994.
14. S. Han, K. Shin, and H. Rosenberg. “Doctor: An integrated software fault injection environment for distributed real-time systems”, Proc. Computer Performance and Dependability Symposium, Erlangen, Germany, 1995.
15. S. Dawson, F. Jahanian, and T. Mitton. Orchestra: A fault injection environment for distributed systems. Proc. 26th International Symposium on Fault-Tolerant Computing (FTCS), pages 404–414, Sendai, Japan, June 1996.
16. D.T. Stott and al. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In Proceedings of the IEEE International Computer Performance and Dependability Symposium, pages 91–100, March 2000.
17. R. Chandra, R. M. Lefever, M. Cukier, and W. H. Sanders. Loki: A state-driven fault injector for distributed systems. In In Proc. of the Int.Conf. on Dependable Systems and Networks, June 2000.
18. X. Li, R. Martin, K. Nagaraja, T. Nguyen, B.Zhang. “Mendosus: A SAN-based Fault-Injection Test-Bed for the Construction of Highly Network Services”, Proc. 1st Workshop on Novel Use of System Area Networks (SAN-1), 2002
19. N. Looker, J.Xu. “Assessing the Dependability of OGSA Middleware by Fault-Injection”, Proc. 22nd Int. Symposium on Reliable Distributed Systems, SRDS, 2003
20. http://www.lri.fr/~fci/GdX
21. S. Lumetta and D. Culler. “The Mantis parallel debugger”. In *Proceedings of SPDT’96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 118–126, Philadelphia, Pennsylvania, May 1996.
22. G. Fedak, C. Germain, V. Néri, and F. Cappello. “XtremWeb: A generic global computing system”. Proc. of IEEE Int. Symp. on Cluster Computing and the Grid, 2001.
23. I.Foster, C. Kesselman, J.M. Nick and S. Tuecke. “Grid Services for Distributed System Integration”, IEEE Computer June 2002.
24. J. Kephart. “Research Challenges of Autonomic Computing”, Proc. ICSE05, International Conference on Software Engineering, May 2005

Maintaining a structured overlay network in a hostile environment

(Extended Abstract)

Kevin Glynn, Raphaël Collet, and Peter Van Roy

Université catholique de Louvain
1348 Louvain-la-Neuve, Belgium

{glynn,raph,pvr}@info.ucl.ac.be

This extended abstract describes work we are undertaking to improve the robustness of a structured Peer-to-Peer overlay network when deployed on realistic networks such as the Internet. We describe a number of strategies which should simplify its implementation, improve the guarantees we provide to application developers, and provide a more stable environment for deployed applications.

P2PS [1] is a Peer-to-Peer (P2P) networking library we have released for the Mozart/Oz platform [2, 3]. Peers connecting with P2PS form an efficient structured overlay network based on Tango [4], a variant (with better scalability) of the distributed k-ary search trees used in other structured P2P systems such as Chord [5], P-Grid [6], DKS [7], Pastry [8], OpenDHT [9], and CAN [10].

If N is the maximum network size then each peer has a node identifier in the range $\{0 \ldots (N-1)\}$ and peers form a ring with *predecessor* and *successor* connections to the peers with next smallest and next largest identifiers respectively. Additional connections (*fingers*) to other nodes in the network allow efficient message routing and broadcasts.

P2PS directly supports message passing between peers, broadcasting to all peers in the network, multi-casting to a list of peers, and key-based routing (where each peer is responsible for keys between its predecessor and itself). With key-based routing the network can be viewed as a *Distributed Hash Table* (DHT) and work can be load-balanced by hashing it to an identifier in the range $\{0 \ldots (N-1)\}$ and sending it to the responsible peer for processing.

Our CoreGRID partners at KTH/Royal Institute of Technology and the Swedish Institute of Computer Science (SICS) are developing the Distributed K-ary System (DKS) [7, 11] a peer-to-peer network with similar properties to P2PS written entirely in JAVA.

We are using P2PS and DKS as the underlying networks for P2PKit [12], a general, service-oriented framework for developing robust, scalable decentralised applications. P2PKit allows services running on a P2P network to be installed and upgraded dynamically at any time and provides hooks so that the application can adapt to the arrival, removal, or sudden failure of peers running the application.

Not surprisingly, we have great difficulties when we move our distributed applications from the laboratory to the real world. In particular, we are testing

our applications on the PlanetLab network which suffers from poor / erratic response times and regular node unavailability. We found that our network did not always adapt well to these problems. For example, messages are often lost, or endlessly loop between nodes trying to find their destination, the peers have inconsistent routing tables, peers cannot agree about whether a peer has really died or not (since they are using different communication paths), and so on.

The following sections give an overview of the improvements we are making to P2PS to address these problems.

Reliable and Ordered Messaging The underlying message passing facilities of P2PS and DKS are unreliable and unordered. A message sent to another peer will be routed through intermediate peers in order to reach its destination. Even though the messages are sent between peers over a reliable TCP/IP connection there are many reasons why the message might not arrive, for example: intermediate peers may crash, run out of buffer space for the message, or be temporarily unable to route messages due to inconsistent predecessor, successor, or finger table entries. In these cases P2PS simply drops the message.

Messages can also arrive out of order. Intermediate nodes forward messages concurrently, which might change message order, and successive message sends to the same destination can take different paths through the network because the finger table entries are constantly changing as peers adapt to knowledge about the existence / disappearance of other peers.

P2PS builds reliable, ordered message passing on top of the underlying unreliable, unordered messaging primitives by maintaining *virtual* end-to-end connections (c.f. the TCP protocol [13]). Source peers identify each message sent to a particular destination with a sequence number. If the message is received an acknowledgement message is sent back to the sender (acknowledgements are batched before sending and, if possible, piggy-backed on application messages going in the opposite direction). At the receiving end messages are delivered to the application in sequence number order (i.e., sending order). If no acknowledgement is received within a reasonable time limit then the message is re-sent. If the message can not be delivered (e.g., because the destination peer has died) the application is informed.

Having a reliable message send mechanism is essential for most interesting applications. We will extend the existing P2PS implementation to work for messages routed by key, and for message broadcasts and multicasts.

When routing by key the peer responsible for the key may change which makes it tricky to manage the required resends and acknowledgements. We solve this problem by adding additional probe messages which identify the correct responsible peer. These probe messages are re-sent if a reliable message is sent to a peer which is no longer responsible for that key.

Messages broadcast from a peer carry a sequence number. The message is sent to each directly connected peer along with the identifiers of the section of the ring it should deliver the message to. Once a peer has successfully delivered the message to all peers in its range it will reply to the sender with an acknowledgement. When the sender receives the acknowledgement from each of

its directly connected peers the broadcast is completed. If acknowledgements are not received within some time then the message is re-sent as necessary. Of course, each connected peer uses the same algorithm to deliver the message to each of its fingers in the range it is responsible for. In this way we can ensure reliable and ordered broadcasts. This protocol guarantees the useful property that any peer that is in the network when a broadcast is sent and which remains in the network will eventually receive the message.

We will also investigate support for ordering between broadcasts and message sends. For example, if an application broadcasts message A to all peers and then sends message B to a single peer, that peer should deliver message A to the application before delivering message B.

We have a paper in progress which will describe these extensions in more detail.

Ensuring Progress Since finger tables are constantly changing it is possible for finger tables to be temporarily inconsistent which, in practice, can lead to messages looping between peers as they try to reach their destination. We will add a simple test to the routing algorithms in P2PS to ensure that as messages are routed they are always getting *closer* to their target peer at each hop.

Since this test will occasionally fail due to the inconsistent finger tables we include a *discrepancy count* in messages. This works rather like a time to live (TTL) counter in TCP/IP. If we cannot route to a strictly closer peer and the discrepancy count is non-zero we will decrement it by 1 and forward the message to a more distant finger where routing will continue. Only when the discrepancy count reaches zero do we discard it.

We believe that even a small discrepancy count will usefully reduce the number of lost messages when network churn (the rate of peers joining and leaving the network) is high.

Finding a New Successor To maintain a coherent network it is important that the successor and predecessor fingers are correct (if finger table entries are wrong then a message will take more hops than necessary to arrive, but it will eventually reach its target peer). All systems of which we are aware maintain a *successor list* holding the addresses of the next F peers which follow it in the ring (where F is a network constant). In this way, if a peer loses contact with its successor it can try to contact each following peer in turn until it finds a working successor. This mechanism allows the network to be able to regain coherency despite the loss of up to F peers.

The successor list adds considerable complexity to the algorithms, and additional communication overhead in maintaining each peer's successor list (every time a peer joins or leaves the network the successor lists of F peers must be updated). In addition, it limits the network's resilience to multiple failures to an arbitrary value of F.

We will remove the successor lists: peers will search for the correct successor via their remaining fingers. In Tango fingers are placed symmetrically around the ring and locating a peer can involve moving between peers both ahead and

behind the target in the ring (hence the name Tango). This default routing strategy may attempt to go through the peer we are trying to avoid, so we will introduce variant strategies which search for peers only in a clockwise, or anti-clockwise, direction.

This allows a peer to find its real successor (by going to its first finger and searching for the successor anti-clockwise) and its real predecessor (by going to its last finger and searching for the predecessor clockwise).

Currently, when a peer loses contact with its predecessor it waits idly for a new predecessor to contact it. Our new routing algorithms allow the peer to search for a new predecessor and eagerly inform our new predecessor about the problem.

Removing Problematic Peers As we explained earlier it is important that peers in a P2PS network maintain working connections to their predecessor and successor nodes. However, due to the real-world behaviour of networks it can be difficult to make a local decision on the best way of resolving any problems.

For example, consider three successive peers on the ring, A, B, and C. It is possible that peer A cannot communicate with its successor peer B, but that communications between peers A and C, and peers B and C are working well. (This is the *non-transitive* communication problem that has been studied previously [14, 15]). In P2PS peer A will ask C to be its new successor, but peer C will refuse and tell it to connect to peer B. The three peers now disagree and we must resolve the situation.

Initially, we propose that peer A should be responsible for choosing its successor. Peer C will then change its predecessor to be peer A and inform peer B that it should shut down (and possibly rejoin the network later). Peers A and C will also communicate their decision to other peers so that the network will gradually start to ignore messages from peer B in the event that it does not shut down willingly.

Of course, it may be that the network would be better off if peer A had shut down. We will experiment with schemes where a number of neighbours in the ring will coordinate to decide which peer(s) would be best to force out of the network.

Note that, as in DKS, peers have unique *nonce*s which appear in their messages. The nonce is generated afresh every time the peer (re-)joins the network so peers will only freeze out a particular peer id / nonce combination and new peers joining with the same id will be unaffected.

Correction-On-Use / Correction-On-Change Chord maintains finger tables by running a periodic stabilisation process. This adds extra overhead to the system for running the stabilisation and leaves the system in a non-optimal state as nodes join and leave. DKS employs *Correction-on-Use* and *Correction-on-Change* [16] mechanisms to reduce this overhead and improve routing performance.

In Correction-On-Use peers improve their finger tables in response to application messages. When a peer receives a message forwarded from a peer (P) it

can make a finger to P if it would be a better finger than it has already. Also, it will check to see if P could have routed more efficiently by using one of its other fingers, and if so inform P. This is a lazy protocol that has little additional overhead. Peers automatically optimise their finger tables as a result of application messages.

In Correction-On-Change, when peers join or leave the network they advertise this fact to the peers that should be pointing to them. In [16] the authors show that the affected peers can be found efficiently and that Correction-On-Change is effective at maintaining near-optimal message paths.

Currently P2PS only supports Correction-On-Use. We will add support for Correction-On-Change. This should help our peers to have more stable fingers since after peer additions and removals they will immediately change to an optimal configuration, rather than moving slowly to the correct place as a result of application messages.

Summary We believe that by incorporating these improvements in P2PS (and where appropriate DKS) we can significantly improve the reliability of our P2P networks in real-world environments. Work is currently under way to implement these improvements in P2PS and measure their impact.

Acknowledgements This work is supported by the CoreGRID (contract number: 004265) and EVERGROW (contract number: 001935) projects, funded by the European Commission in the 6^{th} Framework programme.

Much of the practical feedback for this work is a result of experiments on the PlanetLab network (see `http://www.planet-lab.org/`). Currently, PlanetLab provides a network of around 600 machines hosted by 275 sites in 30 countries around the world.

References

1. Mesaros, V., Carton, B., Van Roy, P.: P2PS: Peer-to-peer development platform for Mozart. In Van Roy, P., ed.: Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004. Volume 3389 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005)
2. Mozart Consortium: Mozart Programming System Release 1.3.1. `http://www.mozart-oz.org` (2004)
3. Carton, B., Mesaros, V., Glynn, K.: `P2PS`: A peer-to-peer networking library for Mozart/Oz. (`http://p2ps.info.ucl.ac.be/index.html`)
4. Carton, B., Mesaros, V.: Improving the scalability of logarithmic-degree DHT-based peer-to-peer networks. In Danelutto, M., Vanneschi, M., Laforenza, D., eds.: Euro-Par 2004 Parallel Processing. Volume 3149 of Lecture Notes in Computer Science., Springer (2004) 1060 – 1067
5. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: ACM SIGCOMM. (2001)

6. Aberer, K., Cudr'e-Mauroux, P., Datta, A., Despotovic, Z., Hauswirth, M., Punceva, M., Schmidt, R.: P-Grid: a self-organizing structured P2P system. SIGMOD Rec. **32** (2003) 29–33
7. Alima, L.O., El-Ansary, S., Brand, P., Haridi, S.: DKS(N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In: Proc. of the 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems – CCGRID2003, Tokyo, Japan (2003)
8. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware). (2001) 329–350
9. Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I., Yu, H.: OpenDHT: a public DHT service and its uses. In: SIGCOMM '05: Proceedings of the 2005 conference on applications, technologies, architectures, and protocols for computer communications, New York, NY, USA, ACM Press (2005) 73–84
10. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. SIGCOMM Comput. Commun. Rev. **31** (2001) 161–172
11. KTH/SICS: Distributed K-ary System (DKS): A peer-to-peer middleware. (`http://dks.sics.se/index.html`)
12. Glynn, K.: `P2PKit`: A services based architecture for deploying robust peer-to-peer applications. (Universtité catholique de Louvain. `http://p2pkit.info.ucl.ac.be/index.html`)
13. Postel, J.: Transmission Control Protocol. RFC 793 (Standard) (1981) Updated by RFC 3168.
14. Freedman, M.J., Lakshminarayanan, K., Rhea, S., Stoica, I.: Non-transitive connectivity and DHTs. In: Proceedings of USENIX WORLDS 2005. (2005)
15. Gerding, S., Stribling, J.: Examining the tradeoffs of structured overlays in a dynamic non-transitive network (2003) http://pdos.lcs.mit.edu/∼strib/docs/projects/networking_fall2003.ps.
16. Ghodsi, A., Alima, L.O., Haridi, S.: Low-bandwidth topology maintenance for robustness in structured overlay networks. In: 38^{th} International HICSS Conference, Springer (2005)

Service and Resource Discovery Using P2P

Sami Lehtonen, Sami Pönkänen, and Mika Pennanen

VTT Information Technology, P.O.Box 1203, FIN-02044 VTT, FINLAND,
Sami.Lehtonen@vtt.fi

Abstract. In the near future, the way of using the Internet is changing. The Internet and services are available for millions of small mobile terminal devices - with different kind of hardware e.g. processor, memory and bandwidth than current desktop computers. Additionally, different network technologies make things a bit more complicated - particularly because of heterogeneous networks and devices. The heterogeneous networks, terminal devices, and their user interface raise new demands on services. Deploying an efficient service and resource discovery mechanism can solve many of these challenges.

1 Introduction

In this paper we introduce Boris Object Request InfraStructure (or BORIS for short). First we discuss some technologies and definitions, then we talk about what is BORIS, and describe the design and architecture of BORIS. After this, we show a couple of example scenarios of utilising BORIS (the latter of the examples, is the solution mentioned in the abstract). Last topic is conclusions and future work.

2 Technologies and Definitions

2.1 Naming Concepts

One of the key questions with services is the way of naming them. For services to be searchable or even accessible one must have some kind of a name. In this document we usually speak of resources rather than services. We define a resource to be anything that can be reached or accessed via a network (including - but not limited to - files, documents, and services).

Names should be unambiguous to be useful. For example, there are only one IP address space for each domain name in the Internet. However, there may be (and usually are) many names pointing the same address space. This is fine, since there is still only one address space for each name. Of course, it is also obvious that an IP address always points a single location although that location could be pointed by several IP addresses.

Unified Resource Name (URN) [1] is a global unique name given to resource. URN is a name and just a name, it does not tell the location of the resource or

the way the resource can be reached. It merely tells that there exists a resource with the given name.

Unified Resource Location (URL) [2] is link that tells the location and reachability information of a resource. In technical words URL tells what protocol to use and what parameters are needed. Note that a resource may be reached in many ways. Thus a single URN might have multiple URL's associated with it. The URL might not even be static due to the dynamic nature of current networks.

2.2 Peer-to-peer Technology

Let's have a definition by Ross Lee Graham:

"In pure peer-to-peer (P2P) there is no central server. Every node is a Peer. It is a total democratisation of the peer group nodes. There are two general forms this architecture may take depending on how the routing is achieved:

- One possible routing structure is the distributed catalog. The router function, using indexes as parameters, searches a distributed catalog. This requires a dynamically balanced structure (to maintain equality for the memory burden among the peers).
- Another possible structuring is direct messaging which is relayed throughout the Peer group members until the object of the inquiry is found or until it is determined that no member of the horizon group has it. ('Horizon' indicates the limit of visibility from the node generating the query, etc.) Some implementations require preset limits to this visibility, others do not.

Whatever search method implemented is managed by the peer group management system (P2PMS) that is cloned for the equality of each peer. In fact, it is the status of a particular installed operational P2PMS that defines the membership status with respect to the group." [3]

P2P systems can be divided in two groups: hybrid P2P and pure P2P. In pure P2P systems, all peers are equal. In hybrid P2P only some of the nodes are router nodes. Thus, all peers are not equal. Boris is clearly a hybrid P2P system which will be shown in the Architecture and Design.

3 What is BORIS?

The main objectives of BORIS was to design a lightweight communications infrastructure that is fast, effective and operates even with limited resources (memory, CPU), and supports mobility. In other words, BORIS is designed to be suitable for Ambient Intelligence and Residential Networking applications. Information is distributed among different devices with P2P technology. Thus, there is no central server dependency. P2P technology also provides means to self-organise the system. The BORIS implementation in each device can be tailored according to the characteristics and resources of that node.

Many questions have been asked about BORIS and it's differences to existing technologies such as Jini or CORBA. The following paragraphs shed light on what BORIS is and what it isn't. BORIS is not a platform in the same sense Jini is. Jini allows Jini services to discover, search, find and interact with other Jini services. All interaction is handled by Jini via Jini Service Proxies. Corba has it's own counterparts called Stubs and Skeletons, and communication is handled by the IIOP -protocol.

In contrast BORIS merely offers sophisticated resource discovery and naming/trader services and allows inter resource communication to be implemented by other means. This means that BORIS does not have any dependencies to any implementation language, design philosophy or transport protocol.

BORIS is not RMI (Java Remote Method Invocation or Corba Remote Invocation). Applications using BORIS do not use object references to invoke methods of remote resources. Rather they use URL's given by BORIS to contact remote resources.

BORIS is not just a metadata search engine. Google and others do a great job in offering powerful search facilities in a general and simple way. While BORIS in no way competes with web search engines, it tries to offer the same kind of attribute-based search service. However there are few important differences.

- BORIS is decentralised, meaning that BORIS nodes have a capability to automatically discover other BORIS nodes and form dynamic rings of distributed databases.
- BORIS does Attribute-to-URN searches and URN-to-URL translations. This split-up allows separation of existence and reachability.

4 Design and Architecture

4.1 BORIS Architecture

The concept of BORIS is split up in three different types of BORIS components:

- TINY Stripped implementation. Capable to register resources to a FAT. Optionally able to make queries.
- SLIM Midi implementation. In addition to TINY, SLIM is able to forward Boris requests and replies.
- FAT Full implementation with information and location database.

BORIS has two services. Metadata Information Service (MEDI) handles resource registration and searches based on resource identity and characteristics. It takes attributes as input and returns a list of URN's as output. The second service is Boris Naming Service (BONA). The naming service provides applications with immunity to network resource location changes (which is essential in dynamic environments). It allows 'on-the-fly' associations between application entities and network resources. It takes a URN as input and returns a list of URL's as output.

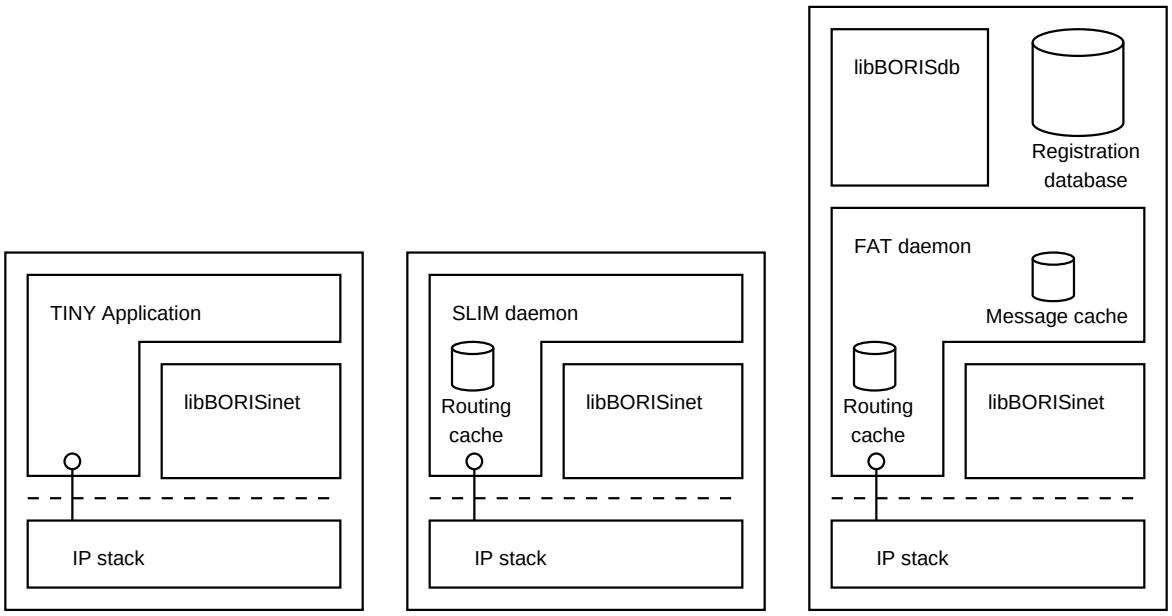


Fig. 1. BORIS Architecture

Since BORIS is designed to be lightweight and suitable for i.e. sensor networks, the Metadata is not replicated across the peer network. The most common requirement in service discoveries is the vicinity of the service and therefore the needed service is the closest one. There are several viable ways of implementing the P2P backbone between FATs (one very interesting being chord [4] by MIT).

4.2 Service attributes

Attributes in BORIS registrations and MEDI queries are presented in the following format:

$$keyword = value \tag{1}$$

Different objects require different kind of keywords. However it is not feasible or even possible to include all keywords that describe an object. An adequate subset of all keywords has to be defined in order to achieve queries that are at the same time fast and specific.

Attribute searches are performed on the given keyword and the subtree below it. Thus a search with no keyword (null) leads to a search of all attributes. Similarly a search on keyword 'name' goes through keywords 'name', 'filename', and 'servicename'.

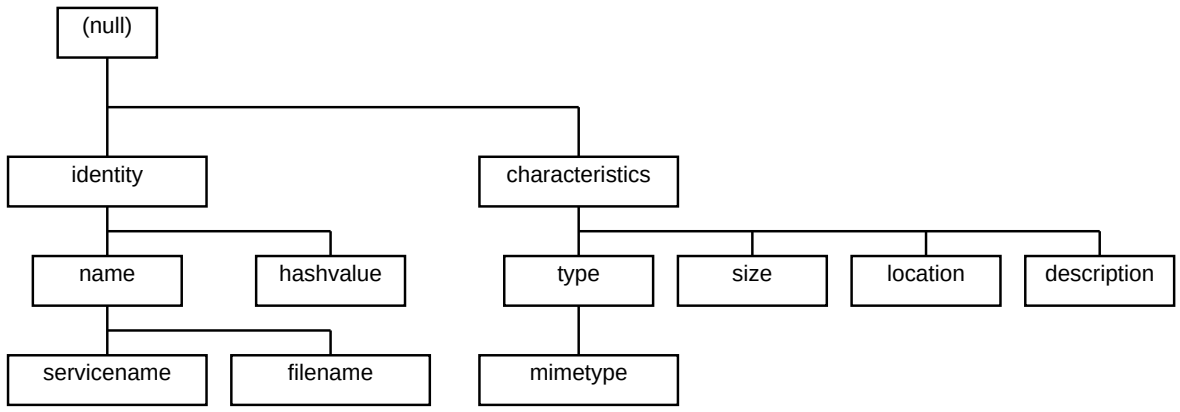


Fig. 2. BORIS Attribute Hierarchy

5 Example Scenarios

5.1 URN's In Web Browsing

The first scenario is very simple. It just shows how the division of attribute to URL resolution in two phases works. The advantage of this approach is that BORIS could be used to solve obsolete link problem (which seems to get worse every day) in web pages.

First, the web browser does a MEDI request according to user (human or application) given attributes:

$$author == "Sami\ Lehtonen"\ \&\&\ name == "Master's\ Thesis" \tag{2}$$

P2P ring of FATs will reply with a URN or list of URNs:

$$URN : nbn : fi - fe20031200 \tag{3}$$

After getting the URN, browser asks for a location with a BONA request. The browser gets an ordinary (or a list of) URL as a reply.

$$URL : http : //edu.lut.fi/LutPub/web/nbnfi - fe20031200.pdf \tag{4}$$

Although this scenario was not in mind when the BORIS was designed it only proves that it may be used (either manually or automatically) in many common situations when communicating in the network.

5.2 Service Naming Broker

In active networks there is the problem of finding a particular service for a particular problem. In the LANE [5] architecture the user just 'has' some services by default. This example scenario introduces a solution that utilises BORIS as the service naming broker.

The scenario exploits the separation of names and locations. An efficient broker will resolve a particular resource by its name rather than location. This serves the need for context-aware functioning, because location is strongly related to context. A resource may be defined for example by a Web Service Description (WSD) which contains information for service selection decision.

The task for the Service Naming Broker is to map attributes to names and names to locations by utilising a local name service database. This local database must be self-organising and self-configuring so that the network is able to behave in ad-hoc manner and still hold as a basis for context-aware decisions.

As a simple example would be a need for printing. This results in searching a printing service or printer resource in the nearby area of the user. A printer or an equivalent resource must bring itself (TINY implementation) in common knowledge through the local name service database (any FAT nearby) and publish its WSD and location information. The network behaviour does not make any difference between mobile and fixed resources.

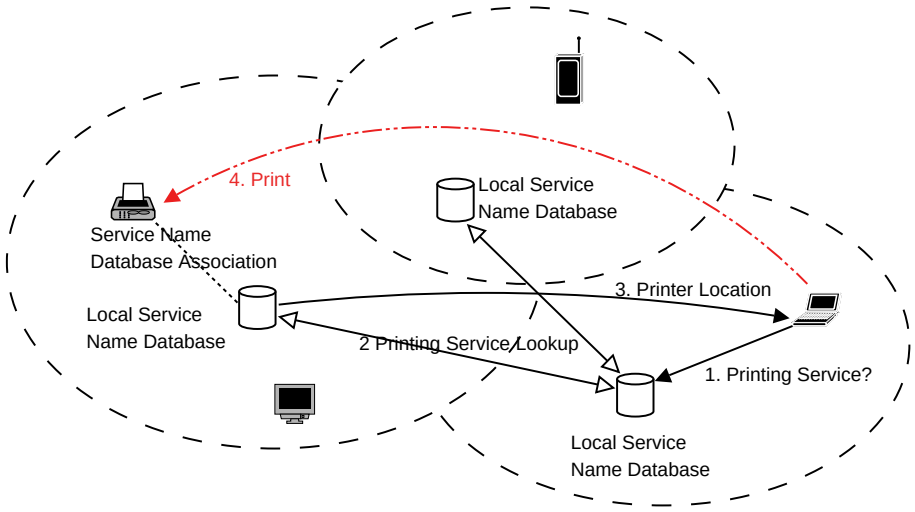


Fig. 3. Service Broker printing example scenario

In this technical area the usage of URNs has a rising trend. It should be taken into account when developing a Naming Broker which can map names to locations. This is essential as URLs - the current de-facto - are usually interdependent with means of communication.

This Naming Broker would benefit if the local resource name database is implemented in a peer-to-peer like structure. This means, that the closest resources store their information into immediate neighborhood. It also supports the flexibility and the ad-hoc behaviour of the system. The way this database works would be a simple distributed pull.

The Pervasive Service Naming Broker could also serve as a policy enforcement point. It is an ideal location for policy rules filtering out unnecessary, unwanted, obsolete, and unauthorised resources for particular users or terminals. It may

also restrict the discoveries into different scopes of the network. The scope used in IPv6 broadcasts would be an analogy to this.

6 Conclusions and Future Work

In this paper, we have presented a solution to the problem of resource naming and discovery. Our approach - the BORIS concept - is a lightweight infrastructure that is implemented in a decentralised manner. BORIS provides efficient resource discovery capabilities based on metadata-to-URN and URN-to-URL resolution services. BORIS does not have any dependencies to the means of inter resource communication, nor does it pose any requirements to the implementation language or target platform.

We have shown two example scenarios which demonstrate the use of BORIS. There are many more interesting and promising application areas. For example Smart Objects, Ambient Intelligence, Residential Networks, Ad-Hoc Networking, Smart Cards, and Mobile Applications to name some of these areas.

The first milestone was already reached in July 2003 when we finalised the first BORIS specification draft. The first prototype stack implementation was created using C3PF protocol framework in early 2004 and native C implementation for unices at end of 2004, which was used in the early demonstrator of IST MAGNET [6]. During 2004 and 2005 we have also implemented several BORIS aware applications for demonstration purposes e.g. network pool game [7], P2P-chat, 3D virtual rooms, sensors, file sharing using web-browser.[8]

Our further research focuses on P2P algorithms and aims at making resource searching more efficient in large scale BORIS networks.

References

1. Moats, R.: URN Syntax urn:ietf:rfc:2141 url:ftp://ietf.org/rfc/rfc2141.txt
2. Uniform Resource Locators (URL) urn:ietf:rfc:1738 url:ftp://ietf.org/rfc/rfc1738.txt
3. Graham, R.L.: What is P2P?
   http://www.ida.liu.se/conferences/p2p/p2p2001/p2pwhatis.html
   last accessed 29.09.2005
4. Chord Website
   http://pdos.csail.mit.edu/chord/ last accessed 09.11.2005
5. Lehtonen, S.: Lightning active node engine - an active network user service platform ERCIM News. (2003) 54, s. 35.
6. IST MAGNET homepage http://www.ist-magnet.org/ last accessed 29.09.2005
7. Pennanen, M, Keinänen K: Mobile Gaming with Peer-to-Peer Facilities Ercim News. (2004) 57, s. 31.
8. VTT BORIS Demonstration
   http://www.vtt.fi/tte/tte33/demos/boris/index.html last accessed 29.09.2005

Self Management of Large-Scale Distributed Systems by Combining Structured Overlay Networks and Components*

Peter Van Roy[1], Ali Ghodsi[2], Jean-Bernard Stefani[3] Seif Haridi[2], Thierry Coupaye[4], Alexander Reinefeld[5], Ehrhard Winter[6], and Roland Yap[7]

[1] UCL/Universit catholique de Louvain, pvr@info.ucl.ac.be,
[2] KTH/Royal Institute of Technology, {aligh, haridi}@kth.se,
[3] INRIA/Institut National de Recherche en Informatique, Jean-Bernard.Stefani@inria.fr
[4] France Telecom R&D, thierry.coupaye@francetelecom.com
[5] ZIB/Zuse Institute in Berlin, ar@zib.de
[6] E-Plus Mobilfunk, Ehrhard.Winter@eplus.de
[7] NUS/National University of Singapore, ryap@comp.nus.edu.sg

Abstract. This position paper envisions making large-scale distributed applications self managing by combining *component models* and *structured overlay networks*. A key obstacle to deploying large-scale applications running on Internet is the amount of management they require. Often these applications demand specialized personnel for their maintenance. Making applications self-managing will help removing this obstacle. Basing the system on a structured overlay network will allow extending the abilities of existing component models to large-scale distributed systems. Structured overlay networks provide guarantees for efficient communication, efficient load-balancing, and self-manage in case of joins, leaves, and failures. Component models, on the other hand, support dynamic configuration, the ability of part of the system to reconfigure other parts at run-time. By combining overlay networks with component models we achieve both low-level as well as high-level self-management. We will target multi-tier applications, and specifically we will consider three-tier applications using a self-managing storage service.

1 Introduction

Multi-tier applications are the mainstay of industrial applications. A typical example is a three-tier architecture, consisting of a client talking to a server, which itself interfaces with a database (see Figure 1). The business logic is executed at the server and the application data and meta data are stored on the database. But multi-tier architectures are brittle: they break when exposed to stresses such as failures, heavy loading (the "slash-dot effect"), network congestion, and changes in their computing environment. This becomes especially cumbersome for large-scale systems. Therefore, cluster-based solutions are employed where

* This collaborative work is supported by the Network of Excellence CoreGRID (contract no. 004265), funded by the EU in the sixth framework programme.

the three-tier architecture is duplicated within a cluster with high speed interconnectivity between tightly coupled servers. In practice, these applications require intensive care by human managers to provide acceptable levels of service, and make assumptions which are *only* valid within a cluster environment, such as perfect failure detection.

Lack of self-management is not only pervasive in multi-tier architectures, but a problem in most distributed systems. For example, deploying a distributed file system across several organizations requires much manual configuration, as does adding another file server to the existing infrastructure. If a file server crashes, most file systems will stop functioning or fail to provide full service. Instead, the system should reconfigure itself to use another file server. This desirable behavior is an example of self management.

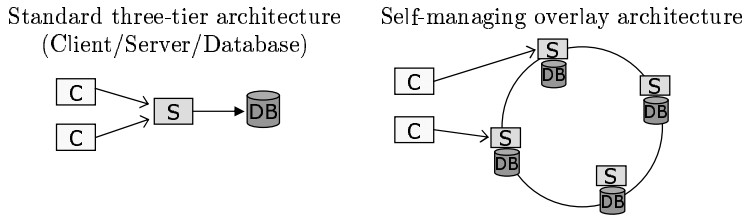


Fig. 1. Left: Traditional three-tier arch. Right: A self-managing overlay arch.

In our vision, we intend to make large-scale distributed applications such as these self managing. In this position paper we outline our vision of a general architecture which combines research on structured overlay networks together with research on component models. These two areas each provide what the other lacks: structured overlay networks provide a robust communications infrastructure and low-level self-management properties for Internet-scale distributed systems, and component models provide the primitives needed to support dynamic configuration and enable high-level self-management properties.

1.1 Definition of self management

Self management and self organization are overloaded terms widely used in many fields. We define self-management along the same lines as done in [1], which can be summarized in that the system should be able to reconfigure itself to handle changes in its environment or requirements without human intervention but according to high-level management policies. It is important to give a precise definition of self management that makes it clear what parts can be handled automatically and what parts need application programmer or user (system administrator) intervention. The user then defines a self management policy and the system implements this policy. Self management exists on all levels of the system. At the lowest level, self management means that the system should be able to automatically handle frequent addition or removal of nodes, frequent failure of nodes, load balancing between nodes, and threats from adversaries. For large-scale systems, environmental changes that require some recovery by

the system become normal and even frequent events. For example, failure becomes a normal situation: the probability that at a given time instant some part of the system is failed approaches 1 as the number of nodes increases. At higher levels, self management embraces many system properties. For our approach, we consider that these properties are classified in four axes of self management: *self configuration*, *self healing*, *self tuning*, and *self protection*.

To be effective, self management must be designed as part of the system from its inception. It is difficult or impossible to add self management a posteriori. This is because self management needs to be done at many levels of the system. Each level of the system needs to provide self management primitives ("hooks") to the next level.

The key to supporting self management is a service architecture that is a framework for building large-scale self-managing distributed applications. The heart of the service architecture is a component model built in synergy with a structured overlay network providing the following self-management properties:

1. *Self configuration*: Specifically, the infrastructure provide primitives so that the service architecture will continue to work when nodes are added or removed during execution. We will provide primitives so that parts of the application can be upgraded from one version to another without interrupting execution (online upgrade) . We will also provide a component trading infrastructure that can be used for automating distributed configuration processes.
2. *Self healing*: The service architecture will provide the primitives for continued execution when nodes fail or when the network communication between nodes fails, and will provide primitives to support the repair of node configurations. Specifically, the service architecture will continue to provide its basic services, namely communication and replicated storage, and will provide resource trading facilities to support repair mechanisms. Other services are application-dependent; the service architecture will provide the primitives to make it easy to write applications that are fault-tolerant and are capable of repairing themselves to continue respecting service level agreements.
3. *Self tuning*: The service architecture will provide the primitives for implementing load balancing and overload management. We expect that both load balancing and online upgrade will be supported by the component model, in the form of introspective operations (including the ability to freeze and restart a component and to get/set a component's state).
4. *Self protection*: Security is an essential concern that has to be considered globally. In a first approximation, we will consider a simple threat model, in which the nodes of the service architecture are considered trustworthy. We can extend this threat model with little effort for some parts, such as the structured overlay network, for which we already know how to protect against more aggressive threat models, such as Sybil attacks.

An essential feature of self management is that it adds feedback loops throughout the system. A feedback loop consists of (1) the detection of an anomaly, (2)

the calculation of a correction, and (3) the application of the correction. These feedback loops exist within one level but can also cross levels. For example, the low level detects a network problem, a higher level is notified and decides to try another communication path, and the low level then implements that decision. Because of the feedback loops, it is important that the system behavior converges (no oscillatory, chaotic, or divergent behavior). In the future, we intend to model formally the feedback loops, to confirm convergent behavior (possibly changing the design), and to validate the model with the system. The formal model of a computer system is generally highly nonlinear. It may be possible to exploit oscillatory or chaotic behavior to enhance certain characteristics of the system. We will explore this aspect of the feedback loops.

2 Related Work

Our approach to self management can be considered a computer systems approach. That is, we give a precise definition of self management in terms of computer system properties, namely configuration, fault tolerance, performance, and security. To make these properties self managing, we propose to design a system architecture and the protocols it needs. But in the research community self management is sometimes defined in a broader way, to touch on various parts of artificial intelligence: learning systems, swarm intelligence (a.k.a. collective intelligence), biologically-inspired systems, and learning from the immune system[1]. We consider that these artificial intelligence approaches are worth investigating in their own right. However, we consider that the computer systems approach is a fundamental one that has to be solved, regardless of these other approaches.

Let us characterize the advantages of our proposed architecture with respect to the state of the art in computer systems. There are three areas to which we can compare our approach:

1. *Structured overlay networks and peer-to-peer systems.* Current research on overlay networks focuses on algorithms for basic services such as communication and storage. The reorganizing abilities of structured overlay networks can be considered as low-level self management. We extend this to address high-level self management such as configuration, deployment, online updating, and evolution, which have been largely ignored so far in structured overlay network research.
2. *Component-based programming.* Current research on components focuses on architecture design issues and not on distributed programming. We extend this to study component-based abstractions and architectural frameworks for large-scale distributed systems, by using overlay networks as an enabler.
3. *Autonomic systems.* Most autonomic systems focus on individual autonomic properties, specific self-managed systems, or focus on specific elements of autonomic behavior. Little research has considered the overall architectural implications of building self-managed distributed systems. Our position is unique in this respect, combining as it does component-based system construction with overlay network technology into a service architecture for large-scale distributed system self management.

We now present these areas in more detail and explain where the contribution of our approach fits.

2.1 Structured overlay networks and peer-to-peer systems

Research on peer-to-peer networks has evolved into research on structured overlay networks, in particular on Distributed Hash Tables (DHTs). The main differences between popular peer-to-peer systems and structured overlay networks are that the latter provide strong guarantees on routing and message delivery, and are implemented with more efficient algorithms. The research on structured overlay networks has matured considerably in the last few years[2–4]. Hardware infrastructures such as PlanetLab have enabled DHTs to be tested in realistically harsh environments. This has led to structured peer-to-peer communication and storage infrastructures in which failures and system changes are handled gracefully.

At their heart, structured overlay networks enable the nodes in a distributed system to organize themselves to provide a shared directory service. Any application built on top of an overlay can add information to this directory locally, which immediately results in the overlay system distributing the data onto the nodes in the system, ensuring that the data is replicated in case some of the nodes become unavailable due to failure.

The overlay guarantees that any node in the distributed system can access data inserted to the directory efficiently. The efficiency, calculated as the number of reroutes, is typically $log_k(N)$, where N is the number of nodes in the system, and k is a configurable parameter. The overlay makes sure that the nodes are interconnected such that data in the directory always can be found. The number of connections needed vary in different system, but are typically in the range $O(1)$ to $O(\log N)$, where N is the number of nodes in the overlay.

Though most overlays provide a simple directory, other abstractions are possible too. More recently, a relational view of the directory can be provided[5], and the application can use SQL to query the relational database for information. Most ordinary operations, such as selection, projection, and equi-joins are supported.

All structured overlays provide self-management in presence of node joins and node departures. This means that a running system will adapt itself if new nodes arrive or if some nodes depart. Self-management is done at two distinct layers: the communication layer and the storage management layer.

When nodes join or leave the system, the communication layer of the structured peer-to-peer system will ensure that the routing information present in the system is updated to adapt to these changes. Hence, routing can efficiently be done in presence of dynamism. Similarly, the storage management layer maintains availability of data by transferring data which is stored on a departing node to an existing node in the system. Conversely, if a new node arrives, the storage management layer moves part of the existing data to the new node to ensure that data is evenly distributed among the nodes in the system. Hence, data is self-configured in presence of node joins and leaves.

In addition to the handling of node joins and leaves, the peer-to-peer system self-heals in presence of link failures. This requires that the communication layer can accurately detect failures and correct routing tables accordingly. Moreover, the communication layer informs the storage management layer such that data is fetched from replicas to restore the replication degree when failures occur.

Much research has also been conducted in making peer-to-peer systems self-tuning. There are many techniques employed to ensure that the heterogeneous nodes that make up the peer-to-peer system are not overloaded[6]. Self-tuning is considered with respect to amount of data stored, amount of routing traffic served, and amount of routing information maintained. Self-tuning is also applied to achieve proximity awareness, which means that routing done on the peer-to-peer network reflects the latencies in the underlying network.

Lately, research has been conducted in modeling trust to achieve security in large-scale systems[7]. In essence, a node's future behavior can be predicted by judging its previous behavior. The latter information can be acquired by regularly asking other nodes about their opinion about other nodes.

2.2 Component-based programming

The main current de-facto standards in distributed software infrastructures, Sun's J2EE, Microsoft .Net, and OMG CORBA, provide a form of component-based distributed programming. Apart from the inclusion of publish-subscribe facilities (e.g. the JMS publish-subscribe services in J2EE), support for the construction of large-scale services is limited. Management functions are made available using the traditional manager agent framework [8] but typically do not support online reconfiguration or autonomous behavior (which are left unspecified). Some implementations (e.g. JBoss) have adopted a component-based approach for the construction of the middleware itself, but they remain limited in their reconfiguration capabilities (coarse-grained, mostly deployment time, no support for unplanned software evolution).

Component models supported by standard platforms such as J2EE (the EJB model) or CORBA (the CCM model) are non-hierarchical (an assemblage of several components is not a component), and provide limited support for component introspection and dynamic adaptation. These limitations have been addressed in work on adaptive middleware (e.g. OpenORB, Dynamic TAO, Hadas, that have demonstrated the benefits of a reflective component-based approach to the construction of adaptive middleware). In parallel, a large body of work on architecture description languages (e.g. ArchJava, C2, Darwin, Wright, Rapide, Piccola, Acme or CommUnity) has shown the benefits of explicit software architecture for software maintenance and evolution. The component models proposed in these experimental prototypes, however, suffer from several limitations:

1. They do not allow the specification of component structures with sharing, a key feature required for the construction of software systems with resource multiplexing.
2. They remain limited in their adaptation capabilities, defining, for those that do provide such capabilities, a fixed meta-object protocol that disallows var-

ious optimizations and does not support different design trade-offs (e.g. performance vs. flexibility).
3. Finally, and most importantly, they lack abstractions for building large distributed structures.

Compared to the current industrial and academic state of the art in component-based distributed system construction, our approach intends to extend a reflective component-based model that subsumes the capabilities of the above models (it caters to points (1) and (2)) in order to address point (3).

2.3 Autonomic systems

The main goal of autonomic system research is to automate the traditional functions associated with systems management, namely configuration management, fault management, performance management, security management and cost management [8]. This goal is becoming of utmost importance because of increasing system complexity. It is this very realization that prompted major computer and software vendors to launch major R&D initiatives on this theme, notably, IBM's Autonomic Computing initiative and Microsoft's Dynamic Systems initiative.

The motivation for autonomic systems research is that networked environments today have reached a level of complexity and heterogeneity that make their control and management by human administrators more and more difficult. The complexity of individual elements (a single software element can literally have thousands of configuration parameters), combined with the brittleness inherent of today's distributed applications, makes it more and more difficult to entertain the presence of a human administrator in the "management loop". Consider for instance the following rough figures: One-third to one-half of a company's total IT budget is spent preventing or recovering from crashes, for every dollar used to purchase information storage, 9 dollars are spent to manage it, 40% of computer system outages are caused by human operator errors, not because they are poorly trained or do not have the right capabilities, but because of the complexities of today's computer systems.

IBM's autonomic computing initiative, for instance, was introduced in 2001 and presented as a "grand challenge" calling for a wide collaboration towards the development of computing systems that would have the following characteristics: self configuring, self healing, self tuning and self protecting, targeting the automation of the main management functional areas (self healing dealing with responses to failures, self protecting dealing with responses to attacks, self tuning dealing with continuous optimization of performance and operating costs). Since then, many R&D projects have been initiated to deal with autonomic computing aspects or support techniques. For example, we mention the following projects that are most relevant to our vision: the recovery-oriented computing project at UC Berkeley, the Smartfrog Project at HP Research Labs in Bristol, UK, and the Swan project at INRIA, Alcatel, France Telecom. Compared to these projects, the uniqueness of our approach is that it combines structured overlay

networks with component models for the development of an integrated architecture for large-scale self-managing systems. Each complements the other: overlay networks support large-scale distribution, and component models support reconfiguration. None of the aforementioned projects provide such a combination, which gives a uniform architectural model for self-managing systems. Note also that many of the above-mentioned projects are based on cluster architectures, whereas our approach targets distributed systems that may be loosely coupled.

3 Synergy of Overlays and Components

The foundation of our approach is to combine a structured overlay network with a component model. Both areas have much matured in recent years, but they have been studied in isolation. It is a basic premise of our approach that their combination will enable achieving self-management in large-scale distributed systems. This is first of all because structured overlay networks already have many *low-level* self-management properties. Structured overlay network research has achieved efficient routing and communication algorithms, fault tolerance, handling dynamism, proximity awareness, and distributed storage with replication. However, almost no research has been done on deployment, upgrading, continuous operation, and other *high-level* self-management properties.

We explain what we mean with lack of high level self-management in overlay networks by the following concrete problems. An overlay network running on thousands of nodes will occasionally need software upgrade. How can a thousand node peer-to-peer system, dispersed over the Internet, be upgraded on the fly without interrupting existing services, and how do we ensure that it is done securely. How can it be guaranteed that the new version of the overlay software will not break when deployed on a node which does not have all the required software. For example, the new version might be making calls to certain libraries which might not be available on every node.

To continue the example, nodes in the overlay might provide different services or may run different versions of the services. For instance, an overlay might provide a rudimentary routing service on every node. But it might be that high-level services, such as a directory service, do not exist on every node. We need to be able to introspect nodes to find out such information, and, if permitted, install the required services on the remote machine at runtime. Even if the nodes do provide a directory service, it might be of different incompatible versions. For example, a node might be running an old version which stores directory information in memory, while another node has support for secure and persistent data storage.

The above mentioned research issues have been ignored by the peer-to-peer community. By using components, we can add these high-level self-management properties, such as deployment, versioning, and upgrade services. Recent research on component models, such as the Fractal model[9], is adding exactly those abilities that are needed for doing self management (such as reification and reflection abilities).

3.1 A Three-tier e-commerce Application

We now give a motivational example which will show how the overlay and the component model is used to build a scalable fault-tolerant application.

Imagine an e-commerce application, which allows users to use their web browser to buy books. The user can browse through the library of books, and add/remove books to its shopping cart. When the user has finished shopping, it can decide to either make a purchase or cancel it.

Traditionally, the above application is realized by creating a three-tier architecture, where the client makes request to an application server, which uses a database to store session information, such as the contents of the shopping cart.

In our system (see Figure 1), there will be several application servers running on different servers, possibly geographically dispersed running on heterogeneous hardware. Each application server is a node in a structured overlay network and can thus access the storage layer, which is a distributed hash table provided by the overlay. The storage layer has a thin layer which provides a relational view of the directory, allowing SQL queries, and supports transactions ontop of the distributed hash table. A user visiting the shopping site will be forwarded by a load-balancer to an appropriate server which can run the e-commerce application. The component model will enable the load-balancer to find the server which has the right contextual environment, e.g. with J2EE installed and with certain libraries, and which is not overloaded. Thereafter the request is forwarded to the appropriate server, which uses the overlay storage layer to store its session state.

To continue our above example, we would like the e-commerce application to be self-healing and provide *failover*. This can be realized by providing a failover component which periodically checkpoints by invoking an interface in the e-commerce application forcing it to save its entire state and configuration to the overlay storage . Should the application server crash, the failure detectors in the crashed node's neighborhood will detect this. One such neighbor is chosen by the overlay and the application is executed on that node. The component model ensures that the last saved state will be loaded by making calls to a standard interface in e-commerce application which will load the session state.

We might want our application to be self-tuning, such that the e-commerce application running on an overloaded server is migrated to another application server . This could be solved using different approaches. One approach would be to have a component which saves the state of a session, and initiates another server to start the e-commerce application with the saved state. Notice that the level of granularity is high in this case as the component model would only define interfaces for methods which the e-commerce application would implement. These methods would then save the application specific state to the storage layer. Similarly, interfaces would be defined to tell the application to load its state from the storage layer. Another approach, with a low-level of granularity, would be to use a virtual machine such as Xen, or VMWare. With these, the whole e-commerce application, its OS and state, would be moved to another machine. This would nevertheless require that the application is running on a common distributed file system, or is getting its data from a common database.

The overlay could be used to either provide a self-managing distributed file system, or let the application use the overlay storage to fetch and store its data. The virtual machine approach has the additional advantage that it guarantees that applications running on the same machine are shielded securely from each other. At the same time, the virtual machine approach would not be able to run if the servers actual hardware differ.

4 Conclusions

We have outlined an approach for building large-scale distributed applications that are self managing. The approach exploits the synergy between structured overlay networks and component models. Each of these areas has matured considerably in recent years, but in isolation. Each area lacks the abilities provided by the other. Structured overlay networks lack the deployment and configuration abilities of component models. Component models lack the decentralized distributed structure of structured overlay networks. By combining the two areas, we expect to eliminate both of these lacks and achieve a balanced approach to self management.

References

1. Herrmann, K., Mühl, G., Geihs, K.: Self-management: The solution to complexity or just another problem? IEEE Distributed Systems Online (DSOnline) **6**(1) (2005)
2. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: ACM SIGCOMM 2001, San Deigo, CA (2001) 149–160
3. Rowstron, A., Druschel, P.: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Lecture Notes in Computer Science **2218** (2001)
4. Alima, L.O., Ghodsi, A., Haridi, S.: A Framework for Structured Peer-to-Peer Overlay Networks. In: LNCS post-proceedings of Global Computing, Springer Verlag (2004) 223–250
5. Chun, B., Hellerstein, J.M., H., R., Jeffery, S.R., Loo, B.T., Mardanbeigi, S., Roscoe, T., Rhea, S., Shenker, S., Stoica, I.: Querying at internet scale. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2004) 935–936
6. Karger, D.R., Ruhl, M.: Simple efficient load balancing algorithms for peer-to-peer systems. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, New York, NY, USA, ACM Press (2004) 36–43
7. Aberer, K., Despotovic, Z.: Managing trust in a peer-2-peer information system. In Paques, H., Liu, L., Grossman, D., eds.: Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM01), ACM Press (2001) 310–317
8. Distributed Management Task Force: http://www.dmtf.org (2005)
9. Bruneton, E., Coupaye, T., Leclercq, M., Stefani, V.Q.J.B.: An Open Component Model and Its Support in Java, Lecture Notes in Computer Science. Lecture Notes in Computer Science **3054** (2004)

Mapping "Heavy" Scientific Applications on a Lightweight Grid Infrastructure

Lazar Kirchev[1], Minko Blyantov[1], Vasil Georgiev[1], Kiril Boyanov[1], Ian Taylor[2], Andrew Harrison[2], Stavros Isaiadis[3], Vladimir Getov[3], Natalia Currle-Linde[4]

[1] Institute on Parallel Processing – Bulgarian Academy of Sciences
{lkirchev, mib, vasko, boyanov}@acad.bg
[2] University of Cardiff, UK
{a.b.harrison, ian.j.taylor}@cs.cardiff.ac.uk
[3] University of Westminster, UK
{s.isaiadis, v.s.getov}@wmin.ac.uk
[4] High Performance Computing Center Stuttgart, Germany
linde@hlrs.de

Abstract. In this paper we present the combined architecture of security and resource management and their close interaction with information service in a lightweight multilevel grid system based on high-level middleware. The models for security control and resource provisioning and sharing are to be implemented by closely-connected modules which interact tightly with information service and stand on role-based security model. These services are local to the cluster level but all of them have the ability to interact with the according neighbors in adjacent clusters. It is important to check the performance and functionality parameters of this clustered grid architecture at an early design phase. That is why here we consider the possibility for mapping of typical scientific application scenario on several commodity computing clusters. For the purpose of exemplary mapping we choose an application scenario of molecular dynamics simulation which is a typical data- and computation-intensive asynchronous application. Here is given the technical mapping of this use case to the underlying infrastructure, paying particular attention to the possibility to implement high-level grid-unawareness for the user or application developer, to identify the compatibility of logical system and technological infrastructure used, as well as service and workflow representation.

1 Introduction

Recently it was commonly recognized that lightweight grid middleware is suitable to diverse organizations in size and domains (while this is not always the case with the extensive production grids). After studying different approaches we have created a hierarchical architecture which enables the participation of different organizations in scale and gives them the ability to be a part of one global grid system or create their own grid infrastructure. Such systems encompass diverse geographically distributed platforms between which communication and interaction should be enabled in a

transparent manner. The architecture must provide underlying system services without imposing excessive overhead and lack of functionality. Another key requirement is that most of the grid service features must be supported by commodity workstations with possibility to extend the infrastructure with dedicated servers (e.g. for data, graphical I/O, etc.). Our approach tries to comply with these requirements and is based on the idea that each entity in the system is a service.

In the next two sub-sections, we present an overview of related work concerning the architectural and infrastructure issues as well as user scenarios issues. Further we present the model we have come to in section 2. The section Architecture Components lists and describes each system module, its place in the overall architecture and how it interacts with the surrounding environment. Section 3 presents the system architecture of the different grid services. Further follows a description of the scientific application of molecular dynamics in section 4 and the problem-infrastructure mapping in section 5. The concluding remarks and the planned work are given in the end of this paper.

1.1 Related Work Overview – Grid Architecture

There are plenty of efforts to create robust and scalable systems. ProActive is a Java library for parallel, distributed and concurrent computing [5]. It provides mobility and security in a uniform framework using a reduced set of simple primitives. ProActive masks the specific underlying tools and protocols used by supplying a comprehensive API which simplifies the programming of applications that are distributed on a LAN, on a cluster of PCs, or on Internet Grids. The library is based on an active object pattern, on top of which a component-oriented view is provided. The implementation of our grid system uses Java for system-independence and its components are service-oriented simplifying its managements and component interaction. The results presented in [3, 4, 5, 6, 7] identified for us the basic properties that the system should possess. Non-functional properties: performance, fault tolerance, security, platform independence and functional properties: access to compute resources, job spawning and scheduling, access to resources, interprocess communication, application monitoring. We have designed our system with these functionalities in mind, aiming to keep the system lightweight and simple for integration and management.

The PROGRESS project provides grid-portal architecture for further deployments in different fields of grid enabled applications [8, 9]. It consists of grid-portal environment tools implemented as open source software packages and the PROGRESS HPC Portal testbed deployment. The software architecture consists of middleware (e.g. Globus, Sun Grid Engine) as well as tools and services created within the project workpackages. The communication between these components is enabled through interfaces based on Web Services and these services are distributed within the testbed installation. We have chosen similar architecture for our solution but it differs in a couple of ways. Our system services are invisible to the user and they communicate with each other using predefined interfaces that describe the functionality of each service. The user is presented with interfaces to which the services he or she provides should conform and the grid environment uses these interfaces to interact with services. These interfaces provide the means to configure the service, enter input data, start the service, monitor its execution and status and retrieve the output data. More-

over, services could be combined in order to create new services from existing ones or use meta-services provided from the framework for the same purpose.

The GridARM [2] is a dynamically extensible, scalable and adaptive system in which new protocols and tools can easily be integrated without suffering from system downtime and expensive code reorganization. In contrast to other gird projects which are based on manual brokerage this project provides an automated brokerage system. This automation is required especially for Grid enabled workflows and execution environments where the brokerage process acts as a middle tier between Meta-scheduler and other Grid enabled components like Grid enabled resources and services [2]. The brokerage process is responsible for discovering and allocating suitable resources for the Meta schedulers. As we aim to create a lightweight environment with a resource management service that imposes as little overhead as possible and without the need for constant human interference for tuning its performance, we agree that it is imperative that the design be scalable and autonomous. The introduction of role-based security model in our architecture to be used for access control to resources eases the automatic decision making from Resource Management Service.

1.2 Related Work Overview – Scientific User Scenarios

The aim of developing new infrastructures is not only to enable users to achieve their aims faster, cheaper, and more accurately, but also to allow them to develop new ways of working that would not have been possible before. With these goals in mind, we have defined a number of key issues that user scenarios expose. Arriving at suitable strategies for handling these issues will lead to a clearer, more flexible, extensible and inclusive design.

Grid-Awareness. A major issue is whether the application the user wishes to run is Grid aware or not. From the user's perspective, most would argue, this difference should be transparent. In some cases legacy applications need to be 'gridified' without changing the behaviour from the user's point of view. In other cases they may need to be wrapped entirely, for example as Web services.

User Interface. As Grid scenarios become more sophisticated, the user interfaces must keep pace. These need to cope with various underlying resource/service/workflow description technologies, many of which are still evolving, in order to render grid entities. Interfaces need to be flexible but also intuitive and simple in order to handle different user types (for example 'grid aware' users may wish to define things such as resources to use while others may not). The design of resource description mechanisms therefore needs to take these issues into consideration.

Infrastructure Used. Users may expect differing grid infrastructures depending on the scenario. For example, certain applications may require highly dynamic and distributed discovery environments. Others may require server-centric data repositories. The ability to behave flexibly according to users' needs in this regard is an important aspect of developing a scalable, generic grid environment.

Middleware Used. Many existing applications already rely on a middleware layer that may be grid enabled in some way. Users and developers will be reluctant to dismantle existing capabilities in order to experiment with new technologies. It is impor-

tant therefore to be able to integrate these into an inclusive grid environment, enabling diverse views of a grid to co-exist.

Service Representation. With new Grid technologies moving towards the service oriented paradigm, shared views of service representation need to be developed. Furthermore, while there are existing standards of service representation and communication with a broad base of acceptance (WSDL and SOAP for example), this area is still in an evolutionary phase. Emerging technologies which are either richer or more efficient need to be able to be integrated when they achieve maturity.

Workflow Requirements. Workflow is becoming more and more important in Grid user scenarios, in part due to the adoption of SOA which views the network as discreet entities providing defined services. Understanding the workflow requirements of users' scenarios will help in defining generic mechanisms for describing and implementing them.

Runtime Requirements. The ability to monitor/steer/migrate running applications is paramount in optimising not only application performance, but user performance as well. These requirements become more complex to implement as the underlying distributed topology becomes more complex.

We have elicited a number of user scenarios from projects affiliated with CoreGrid and we have chosen one particular scenario – the Molecular Dynamics Simulation (HLRS) for a number of reasons, including:

- It is both computation and resource intensive.
- It requires workflow
- The user should not have to be conversant with grid technologies
- It requires a sophisticated user interface
- It requires monitoring and steering capabilities

Section 5 describes how we map the scenario to our Grid environment.

2 Service Oriented Architecture

We have called our combined multilevel grid architecture GrOSD (for Grid-aware Open Service Directory). Each participating party in GrOSD is represented by a cluster – its grid system and all clusters are embraced in a global grid. This present us with a hierarchical multilevel architecture as suggested in models presented in [1].

The central idea of our research is to create a lightweight scalable solution with simple and effective modules. We based our model on Service Oriented Architecture paradigm. After going through different designs and similar grid systems we have identified the cornerstone services of our model and its functionalities. The entry point to the system is the grid portal. Both cluster and global layers have their portals and they are much the same in the services they provide. The system services that we have identified are: Security Service/User Management; Resource Manager Service/Task Scheduler; Information Service; Monitoring Service; Accounting Service; Node Service.

Each system service has a cluster- and a global version. Each one performs its duty in the context of a cluster or grid. Cluster system services interact with each other within the cluster and have the ability (if they are a part of a global grid) to interact with the corresponding services from other clusters. Global grid system services pro-

vide their functionalities in terms of a collection of clusters. They interact with each corresponding cluster service to present their results in terms of global grid infrastructure.

The basic functionalities that our system provides are:

- Library of service prototypes, meta service support and code wrappers
- Directory of active services
- Structured service description/advertisement
- User profiling and accounting

Administrative and user GUI will be supported by cluster and grid portals.

In context with the features listed above in our system each resource is accessed through a service and thus represented by a service and a task for execution is a service with the supplied data and selected and reserved resources. This makes the service model fundamental to our architecture.

3 Architecture Components

The Grid/Cluster portals are the corresponding entry points to our grid model. They represent the user with a front-end where he could log into the system if he is successfully authenticated by the proper Security Service – Grid Security Service if he is logging on the grid portal or Cluster Security Service if he is logging on the cluster portal. The portal lists the active services to which the user has access rights and he could browse them or he could browse the repository with inactive services that the grid/cluster provides access to.

Security Service (SS) authenticates users and provides access information for users and services. Other system services could query Security Service for users' rights confirmation and validation. Data used by SS is stored in Information Service and security sensitive data is accessible only from SS. This service is duplicated on cluster and grid level and each controls its domain. Both interact when they need to proved security functionalities between clusters. GrOSD implements a role based security model. In this model, a number of roles are defined in the grid and rights are associated with roles, not with particular users. This approach has the advantage that there is no need to assign rights to every user separately – the user is assigned one or more roles and automatically receives the rights with them.

Resource Management Service (RMS) is responsible for resource discovery (issuing queries to Information Service for the necessary resources matching the user roles), providing functions to monitor jobs' status for which the Monitoring Service (MS) keeps track of, and task scheduling (tasks could be started immediately, reallocated for execution on specific node or planned for later or exact time execution).

Data about services, service descriptions, user information, node status and information is kept in the Information Service (IS) and used from the other system services as Security and Resource Management. Accounting information and usage statistic for services and nodes' resources are stored in Information Service too.

Node Service (NS) is a peace of software that makes a node part of a cluster and thus of the whole grid infrastructure. It keeps track of node's resources and capabilities, interacts with MS to update information, statistic and status for that node. MS

stores the proper data into IS. NS starts the real execution of tasks. RMS allocates tasks to particular nodes based on resource selection algorithm. NS provides functions for other system services to interact with it in order to monitor job execution, suspend or stop execution and find out tasks that have failed and need to be restarted or reallocated.

Monitoring Service (MS) updates status for each service into the Information Service. Monitoring service polls nodes comprising the cluster for their status and they could inform it too if there is a change in the status. The same applies for tasks in execution.

When a user selects a service, a GUI is shown based on the service description (structured XML). It lists the features of the service what it performs, what are the input parameters, output parameters, resource requirements, cost of usage and its status.

4 GRID Application Scenario: Molecular Dynamics Simulation

The problem is to establish a general, generic molecular model that describes the substrate specificity of enzymes and predicts short- and long-range effects of mutations on structure, dynamics, and biochemical properties of the protein. A molecular system includes the enzyme, the substrate and the surrounding solvent. Multiple simulations of each enzyme-substrate combination need to be performed with ten different initial velocity distributions. To generate the model, a total of up to of 3000 (30 variants x 10 substrates x 10 velocity distributions) MD simulations must be set up, performed and analyzed

Each simulation will typically represent 2 ns of the model and produce a trajectory output file with a size of several gigabytes, so that data storage, management and analysis become a serious challenge. Each simulation can typically require 50 processor days for each simulation. These tasks can no longer be performed interactively and therefore have to be automated.

The scientific user requires an application which is user-friendly (requires no specific programming or GRID knowledge) and can deliver and manage the required computing resources within a realistic time-scale. Such an application requires a workflow system with tools to design complex parameter studies, combined with control of job execution in a distributed computer network. Furthermore, the workflow system should help users to run experiments which will find their right direction according to a given criteria automatically.

This case of Molecular Dynamics Simulation has been mapped to SEGL (Science Experimental Grid Laboratory) – a Problem Solving Environment which has been used to solve a wide range of application scenarios in different fields such as statistical crash simulation of cars, airfoil design and power plant simulation. These scenarios have been tackled using extensive computing resources and parallelization. This article describes in more detail the application scenario of Molecular Dynamics simulation.

SEGL is a grid-aware application enabling the automated creation, start and monitoring of complex experiments and supports its effective execution on the GRID. The

user of SEGL does not need to have the knowledge of specific programming language and knowledge about of GRID structure.

SEGL allows the description of complex experiments using a simple graphical language.

The system architecture of the SEGL consists of three main components: the User Workstation (Client), the ExpApplicationServer (Server) and the ExpDBServer (OODB). The system operates according to a Client-Server-Model in which the ExpApplicationServer interacts with remote target computers using a Grid Middleware Service such as UNICORE and SSH. Integration with Globus is planned for the future. The implementation is based on the Java 2 Platform Enterprise Edition (J2EE) specification and JBOSS Application Server. The database used is an Object Oriented Database (OODB) with a library tailored to the application domain of the experiment.

SEGL consists of two main parts: Experiment Designer (ExpDesigner), for the design of the experiment, and the runtime system (ExpEngine).

The control flow level is used for the description of the logical schema of the experiment. On this level the user makes a logical connection between blocks: direction, condition, and sequence of the execution of blocks. Each block can be represented as a simple parameter study. The data flow level is used for the local description of interblock computation processes.

5 Mapping of Scientific User Scenario to GrOSD Infrastructure

Similar scientific scenarios may be performed by GrOSD. The program that performs the molecular modeling will be represented as a set of non-persistent application services in our grid system. In this case a Master-Workers application distribution model will be implemented. The master application service will support two interfaces: the user interface and the interface to the rest of the services. All of these services have to be submitted to a GrOSD portal (either the grid portal or one of the cluster portals) for co-scheduling and execution. Actually, for GrOSD the master and worker services are just services for execution.

Depending on the developer's choice, the overall application may be grid-aware or grid unaware. The difference between these two approaches lies only in the implementation of the master application service – we have GAMs or GUMs, respectively.

A grid-aware (and particularly GrOSD-aware) master (or GAM) supports one more interface – it interfaces the portal. GAM is responsible for the problem decomposition and its granularity, so that it can decompose the problem domain in different number of subdomains. The decomposition task can be done by GAM using only the domain attributes (e.g. domain size and domain structure, represented by the type of parameter studies). However, the application developer may choose to design GAM that negotiates with the portal the actual parameters of the grid environment (cluster-wide or grid-wide ones) prior to making the decomposition decisions. Then (using its GrOSD interface) it submits the corresponding number of tasks as application services providing them with the appropriate metadata. The collection of the results (or the report of their location) is obviously a responsibility of the master service – either GAM or GUM.

Choosing the GrOSD-unaware master (GUM) approach releases the application developer from the necessity to integrate a grid interface in the master code. The master service (no matter whether GAM or GUM is used) offers a simple and intuitive Graphical User Interface, which describes the service's features – what it performs, what input parameters are needed, its resource requirements. The user may choose from the GUI with what combination of parameters (enzyme-substrate combinations and velocity) the experiment should be carried out. It also should be possible to choose a sequence of parameter combinations for consecutive simulations, which are to be performed one after another. The user may further customize the requirements for resources – e.g., choose a greater number of CPUs for execution, or particular nodes, on which the service should be executed. The user may choose whether the service should start execution right now, or be scheduled for a later moment.

The services (both master and workers) are described with metadata tags. This metadata is represented by XML descriptors, which list the requirements of the service – number of CPU needed for execution, amount of memory, disk space, input and output data structures or requirements (e.g. graphical device output, printers, etc.).

In GrOSD the workflow requirements are represented by metaservices and service wrappers, one of the functionalities of which is to support the transfer of a [worker] service's output to another service[s]'s input.

Monitoring of the services being executed (both master and workers) is performed by the Monitoring Service, which may query the nodes, where the job is running, for their status. This monitoring has system functions – for example, in case of failure appropriate measures to be taken. Further, the worker services should be able to report to the master service the work already done – e.g. number of combinations modeled, or percent of the job finished. The master service presents the user with this feedback through the GUI. The execution of a service at a node is monitored by the Node Service.

In GrOSD the simulation application may be executed either on a single cluster (i.e., on the cluster level of the grid), or in multiple clusters (i.e., on the grid level). Considering the system requirements of the simulation, a single cluster may be inadequate for its execution (unless the cluster is very large). In the general case the scientific applications have to be executed at the grid level.

Figure 1. illustrates the sequence of steps. Following is the sequence of steps, performed by GrOSD for execution of the application (the case of GAM is considered, with grid-level execution; the numbers of the consecutive steps are encircled in the diagram, and correspond to the respective numbers in the following paragraph):

First the GRMS loads the master service (1). After the user has entered all parameters in the GUI, the master service performs the necessary decomposition of the domain among the worker services (2) and sends these services to the GRMS (3), which should schedule them for execution. The GRMS contacts the GIS to find available resources for the execution of the tasks (4). The GIS sends requests to all CIS (5) for resources, available in the clusters (a worker service may be running on every node in every cluster, provided that it has the adequate disk space). After the resources are found, every CIS gives the GIS a list of the available resources in the cluster (6). The GIS sends this list to the GRMS (7) and it decides which resources to use. User restrictions such as choice of particular nodes may be used for this decision. Further, the GRMS contacts the CRMS of every cluster, where resources are to be allocated,

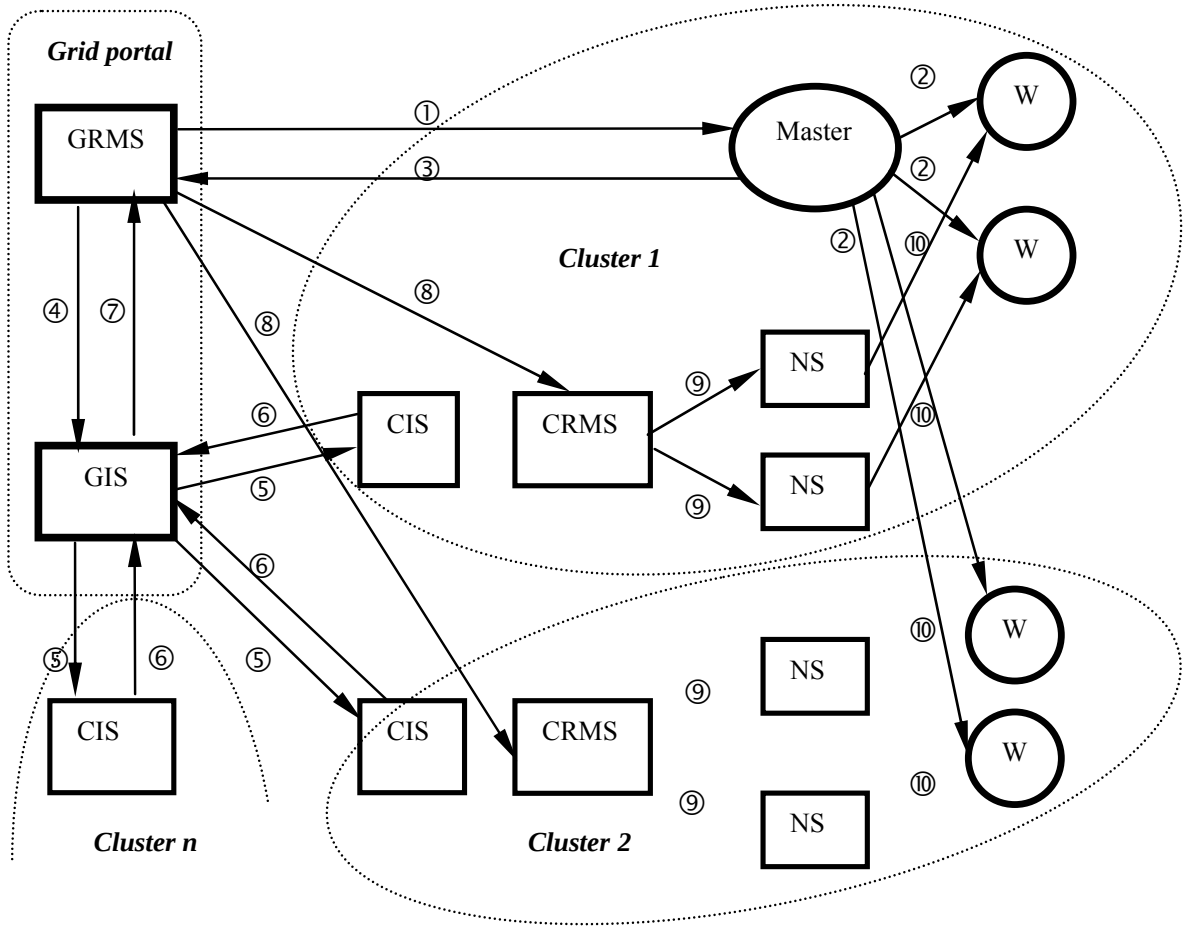


Fig. 1. Application case mapped to the clustered grid

and every CRMS in turn contacts every node in its cluster at which there is a resource, which will be used for the execution of the task. The GRMS sends the worker services, accompanied by the required metadata, to the appropriate CRMS (8), which forwards them to the Node Managers of the nodes (9). The node manager of every node is responsible for running the task (10). When it is finished, the Node Service (NS) should inform the CRMS. Also, the worker service, upon finishing, informs the master that it has finished.

If the execution is in a single cluster, the process of allocation of resources and control of execution will be the same, the only difference being that the services are submitted at a cluster portal. The GRMS and the GIS do not participate in the brokering and allocation, only CRMS and CIS of the cluster are used.

6 Conclusions and Future Work

In this paper we have presented the system architecture of a lightweight multilevel grid system. Our main purpose was to design a scalable and simple for management framework that enables quick and easy establishment of a grid infrastructure from diverse organizations. The Grid and Cluster Portals give local and external users the

opportunity to easily access resources represented by different services, use persistent services shared in the grid or browse service repository for suitable to their needs pre-submitted tasks. Our model provides the ability to create services from existing and meta services. In addition, users could submit their own services.

The multilevel architecture enables sharing resources between different parties spread in diverse geographical locations. Each participant creates his own cluster with its own portal and they could choose to be a part of a global grid system. Underlying system components are implemented as services and have their representation in the cluster and grid layers, thus simplifying the architecture and propagating the system structure through levels. This design eases the resource management activities by providing means to automate the resource discovery and allocation by incorporating role-based security model for controlling the access to services and the use of resources from the services. The whole framework will use Java and Java communication and network technologies like Jini and JXTA. We consider this to be a scalable and platform independent solution.

Our future work is to simulate the work of the whole system and experiment with different algorithms for resource selection and allocation. We will start building a testbed and provide some example services that will give us the chance to examine the properties of our architecture in greater details and provide us with information for future improvements.

References:

1. Buyya, R., S. Chapin, and D. DiNucci. Architectural models for resource management in the Grid. Proceedings of the 1st IEEE/ACM International Workshop on Grid Computing, Bangalore, India, (2000), 18-35.
2. Siddiqui M., Thomas Fahringer, GridARM: Askalon's Grid Resource Management System. Proceedings of the European Grid Conference, Amsterdam, Netherlands (2005),122-131
3. Schwiegelshohn, U., R. Yahyapour, Resource Management for Future Generation Grids, CoreGRID Technical Report, Number TR-0005, (2005)
4. Wieder, P., W. Ziegler, Bringing Knowledge to Middleware – Grid Scheduling Ontology, CoreGRID Technical Report Number TR-0008, (2005)
5. Kielmann, T., Andre Merzky, Henri Bal, Grid Application Programming Environments, CoreGRID Technical Report, Number TR-0003, (2005)
6. Kacsuk, P., N. Podhorszki, Scalable Desktop Grid System, CoreGRID Technical Report, Number TR-0006, (2005)
7. Marios D. Dikaiakos, Rizos Sakellariou, Yannis Ioannidis, Information Services for Large-Scale Grids A Case for a Grid Search Engine, CoreGRID Technical Report Number TR-0009, (2005)
8. Bogdański M., Kosiedowski M., Mazurek C., Wolniewicz M, PROGRESS USE Framework: GRID Service and Access Management within User Service Environment. Presented to the Global Grid Forum, Grid Computing Environments Research Group, September 2002 http://progress.psnc.pl/English/
9. Michał Kosiedowski, Cezary Mazurek, Maciej Stroiński, PROGRESS – Access Environment to Computational Services Performed by Cluster of Sun Systems, Presented at the 2nd Cracow Grid Workshop, Krakow, Poland, (2002) http://progress.psnc.pl/English/

User Profiling for Lightweight Grids

Lazar Kirchev[1], Minko Blyantov[1], Vasil Georgiev[1], Kiril Boyanov[1], Maciej Malawski[2], Marian Bubak[2], Stavros Isaiadis[3], Vladimir Getov[3]

[1] Institute on Parallel Processing – Bulgarian Academy of Sciences
{lkirchev, mblyantov, vasko, boyanov}@acad.bg
[2] Academic Computer Centre CYFRONET AGH - Krakow, Poland
{malawski, bubak}@agh.edu.pl
[3] University of Westminster, UK
{s.isaiadis, v.s.getov}@wmin.ac.uk

Abstract: *User management is important for the effective functioning of a grid system. Here we present a user management model developed for the lightweight multilevel grid architecture. We decided to implement a role based security model, where a number of roles are defined for the grid. Different roles have different access restrictions. They are assigned to users and this is how users receive rights in the grid. Every cluster in the grid has a security service, which handles user's identification and access control. The role based access control allows us to incorporate some simple access decision logic in the information service, which makes controlling user rights easier. Further in this paper we present our user profiling model along with that of the lightweight middleware H2O and MOCCA in order to compare user scenarios, protection functions and technologies.*

1. Introduction

Security is a central issue in computational grids. They are composed of multiple resources and accessed by a large number of users. It is important that not every user has access to every resource, to another user's items, etc.

In this paper the considerations about the user management and security in lightweight grids have been done with respect to the design of the system GrOSD. GrOSD (for Grid-aware Open Service Directory) is a gridware under development and its purpose is the construction of a lightweight grid infrastructure. We set as a main priority the simplicity of the architecture and implementation. Actually this simplicity is a characteristic of the lightweight grid system – vital features of such a system are the ease of deployment, use and maintenance, opposed to the "heavy" production systems, where complex grid middleware such as Globus is used.

The research and design work on GrOSD is going on in the context of the activities of the Virtual Institute on Problem Solving Environment, Tools and GRID Systems

(Work Package 7) of the European Research Network CoreGRID and it is based on consideration of the existing prototypes of the CoreGRID partners. At this stage we take as a pilot prototype the CCA (Common Component Architecture)-compliant MOCCA component framework [15] based on H2O which is referred further in this section.

While analysing the requirements for GrOSD, we defined the following as the most important security requirements for our grid: user management (including user registration and user authentication), authorization for access to resources, data encryption and security of executing code, delegation of rights and, finally, auditing of user access and resources usage.

We have considered different solutions for the above-enumerated issues. There are presently different approaches to grid security. Here we will present some of them.

In the Globus Toolkit [6] most security issues are handled by the Grid Security Infrastructure (GSI) [3, 5]. In Globus there are users, resources and programs. Every entity has a certificate that represents its global identity. The certificate contains the global name of the entity and additional information (e.g., a public key). It is in a standard X.509 format. Verification of identity is done using SSLv3 protocol. It verifies also the identity of the Certification Authority that issued the certificate. In GSI delegation of rights is supported – one entity may delegate its rights (or part of them) to other entities.

An extension to the GSI is the Community Authorization Service (CAS) [16]. CAS makes possible security policy to be enforced on the bases of a global identity, so that mapping to local user accounts is not necessary. The CAS server stores information about who has permission, what permission is granted and which resource is the permission granted on. The resource owners give access to a community (e.g. a Virtual Organization) as a whole, and the community defines the finer rights. When a community member wants access to a resource, s/he sends a request to the CAS server. The server checks if the community policy permits such access. If it does, the server issues a capability that allows the user to perform actions. The user presents this capability to the resource server.

Another solution for authorization, to some extent similar to the CAS server, is the Virtual Organization Management Service (VOMS) [4], implemented in the EU DataGrid project. Every Virtual Organization has a VOMS server, which stores all user information – accounts, rights, groups, and roles.

JGrid [10] is a Java and Jini based computing grid infrastructure, developed by the Veszprem University, Hungary. For user management it uses two services [11, 14]. The Authentication Service is responsible for user authentication and single sign-on. It issues short-term credentials (private key and X.509 certificate) to those users, who have no long-term certificate. The Registration Service stores all user information. It offers role based access control (in which each role represents a permitted actions list), user registration and user management (for the administrators).

EU GridLab [7] is a European research project. GridLab middleware uses a Grid Authorization Service (GAS) [8, 1] for controlling user access. It represents a single logical point for defining security policy. The GAS subsystem comprises an AS (Authorisation Service) Server, database, management module, and has modules for communication with services/applications/users and integration with other security solutions. It has initial support for RBAC (role based access control) security model.

PROGRESS [18] is a project carried out by the Poznan Supercomputing and Networking Centre in cooperation with other institutions. This Grid system [12] has a portal, which serves as a user interface to the grid services. For user identification an Identity Server is used, which authenticates the user and manages sessions. The authentication is done with username and password. For authorization the Resource Access Decision [19] model developed by the OMG is used. The authorization is performed by Resource Access Decision (RAD) Module. The resources and services are classified in a number of types and roles representing specific rights for every type are created. The access rights are associated with the roles. When a user is authenticated, he is issued a token (which usually is the user's session cookie).

H2O [9, 13] is a component-based and service-oriented framework, intended to provide lightweight and distributed resource sharing. It is developed at the Department of Math and Computer Science at Amory University. This architecture is based upon the idea of representing 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.

After considering different solutions, we decided to implement in our grid a role based security model, as in [4, 14, 8]. A Role Based Access Control (RBAC) Security Model is also used in the PERMIS System for user authorization [17]. In this model, a number of roles are defined in the grid and rights are associated with roles, not with particular users. This approach has the advantage that there is no need to assign rights to every user separately – the user is assigned one or more roles and he / she automatically receives the rights with them. We consider this a scalable solution. Moreover, the node manager of every cluster node may impose further restrictions upon the access to the resources, which it controls. In this way an additional flexibility of defining access policy is gained. For security at the level of code execution sandboxing will be used, similar to the approach taken in the AliCE grid system [2, 20]. Thus data and code security are guaranteed by the implementation technology that will be used, namely Java, as is in the H2O project [13]. A detailed comparison between GrOSD and H2O is presented in this paper.

In the rest of the paper we will discuss the following: Section 2 delves upon the user profile management, in Section 3 the user authentication, authorization, data and code security mechanisms are presented. Section 4 elaborates on the comparison of GrOSD and H2O and addresses possible adoption of components from H2O/MOCCA and Section 5 makes the conclusion.

2 User accounts in GrOSD

Before discussing the user profile management in GrOSD, we will make a short overview of the architecture of our grid system. It consists of interconnected clusters of computers. In GrOSD we use the term cluster to denote the unit for organizing nodes, resources and users – it is logical rather than a physical unit. In our system it is

not necessary that nodes in the same cluster reside on the same physical location. The architecture is hierarchical and has three levels. The first level is the local level. At this level are the different clusters, which comprise the system. The second level is made by connecting the clusters – it represents the grid. The last level is the intergrid level. At this level a connection with other grids is made possible. Both the grid level and the cluster level have portals – the grid portal acts as the grid entry point and the portal of every cluster is its entry point. These portals present the user with a front-end for logging into the grid or a particular cluster. The system services are local for every cluster, and also there are respective system services for the grid level. The resources will be presented as services, so that the use of a resource will be actually a use of a service.

Every user in the grid has a personal user account. The account will be unique in the whole grid and every person will have one account – not different local accounts and one global, as is the solution used in other grid systems such as Globus for instance [2]. This user account belongs either to one of the clusters that comprise our grid, or to the grid level, if it is for an external user.

In order to use resources in the grid, the user be required to have an account. So when he/she contacts the grid portal for the first time, the user will be prompted to register so that an account will be created for him/her. In this case the account will be created at the grid level and the user information will be stored at the Grid Information Service. This may be considered a global account, but only external users of the grid will have such accounts. Most users will register at a cluster portal. Then the account that will be created for him/her will belong to the respective cluster and the user information will be stored in the Cluster Information Service. These accounts are local to the cluster, where they are created.

Anyone can apply for an account and the request is sent to an administrator – either a cluster administrator, if the request is for an account in a cluster, or the grid administrator, if the request made at the grid portal. In the request – regardless if the user applies for an account at the grid or cluster portal - the user will include different personal information. The administrator (grid or cluster respectively) decides, on the basis of this information, if an account will be created and what rights the user will have.

Some users may have rights only to access resources that are local to the cluster, where their account belongs. Other users may have rights to access resources in other grid clusters too. In fact, they may access only certain services in other clusters, which are exported by the clusters. The cluster administrator of every cluster decides which services, if any, will be exported by his/her cluster, and thus will be made visible for users from other clusters with sufficient rights. The accounts of these users are still local, but they are given necessary roles that permit such an access.

We may distinguish five types of roles for the grid users: System Grid Administrator; System Cluster Administrator; User with Cluster access; User with Cluster and Grid access and External Grid Users. System grid administrator manages the user accounts at the grid level as well as the roles of the cluster users, which give them access to remote clusters, while every system cluster administrator is responsible only for the accounts in the cluster, which he/she administers. The users with cluster access have rights for the local resources in their cluster, which are defined by a set of roles, kept by the Cluster Information Service. The users with

cluster and grid access have rights both for local resources and for resources in remote clusters. The roles for these users, which define their rights for access of remote resources, are kept by the Grid Information Service. These roles are essentially the same as the local roles, the only difference being that they give access to remote resources. The external grid users are those users, who are not members of any cluster. They have accounts in the grid level. They have no local roles (there are no resources local for them), only roles that define rights for remote resources (since all clusters are remote for them). Thus the rights of each user in the system are defined by the roles, s/he possesses. Fig. 1. illustrates the organization of users and roles.

In the figure, user B has only cluster roles which give him/her access to resources local to his/her cluster. He/she has no rights to access resources outside his/her cluster. On the other hand, user A has, in addition to his/her cluster roles, grid roles. They give him/her access to resources in other clusters. These roles are assigned to him/her by the grid administrator, if the administrator decides that the user really needs remote access and should be granted such access. User E has a global account and has only grid roles. He/she may access resources in different clusters. His/her roles are assigned by the grid administrator.

When a service is published, a part of its description in the Information Service database will be the list of roles, which have access to it, and also a list of roles, which the service needs for execution (in fact, these roles identify what resources the service will use). In this way the owner of a service may define the access level to his/her service. Another feature will be the opportunity to restrict user access locally. For every node in the cluster there will be a node manager, which controls the functioning of the node. For example, it oversees the execution of jobs on the node, sends information about the current status of the node to the information service and so on. It will be possible to make further restrictions on the access to the node's resources by stating which roles may start processes on the node. The node manager will be responsible for enforcement of the restrictions.

A role in our model has descriptive nature. We differentiate several types of resources – e.g. computing, storage, etc. – and for every type we define three roles, with increasing degree of access rights – weak (with least rights), normal and strong (which gives most rights). For example, for the storage resource type we have WeakStorageUsage, NormalStorageUsage and StrongStorageUsage roles. If a user

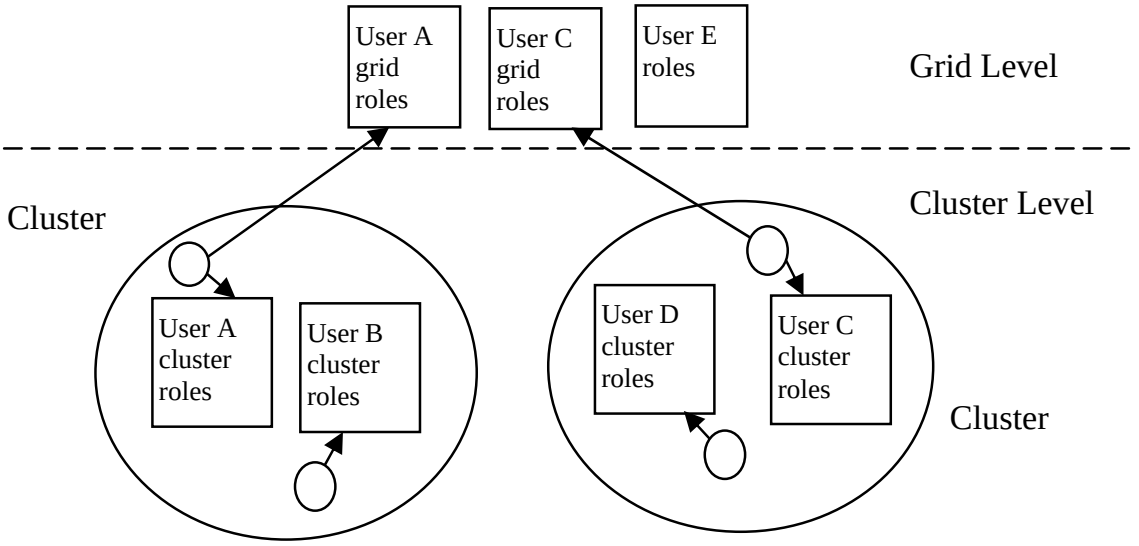


Fig. 1. Users and roles

has the cluster role NormalStorageUsage, the user will be able to use resources in his/her cluster that require Weak or NormalStorageUsage role, but will not be able to use a resource that requires StrongStorageUsage. Possession of a role for one type of resource does not imply possession of the same role for the other types of resources. A user with NormalStorageUsage role does not necessarily has NormalComputeUsage, for example. The role, stored at the cluster level, gives the user the corresponding access rights only for the resources in the cluster. If the role is given to the user for remote clusters (in this case it is stored by the grid information service, and it is given by the grid administrator), the user will have access to all exported services, in all clusters, which require Weak or NormalStorageUsage (if there are no additional restrictions on the resource). The possession of a role in the local cluster does not imply the possession of the same role on grid level. A user may have, for example, locally StrongStorageUsage, but on the grid level he/she may have Weak or NormalStorageUsage for access to remote clusters, or even may have not this role at all. We believe that this is a scalable solution, because the rights for the resources in the different clusters are not given separately to each and every user. The grid roles define the access of users to clusters, other than their local cluster and these roles have the same meaning as the local roles. But if a cluster administrator wishes, he/she may restrict a global role to a weaker role for the same type of resource. This restriction will be valid only in the cluster of this particular administrator. For example, an administrator may decide that in his/her cluster, the global role StrongStorageUsage will be equal (will have the same access rights as) the local role NormalStorageUsage. The administrator may do this in order to limit the storage access for users of remote clusters.

In addition to the individual accounts associated with a particular cluster, there will be a guest account, which will offer anonymous access to the grid. This account will not reside at a particular cluster and will give very restricted rights.

3 Authentication and Authorization

Authentication is the process of proving one’s identity. There are many different methods for authentication. Very often digital certificates are used to authenticate a user, but the method used depends mostly on the security level needed in the grid. We plan to support different mechanisms for authentication – at the beginning we will maybe use a simple username-password authentication and extend the functionality in the future so that it supports digital certificates and other methods. Here it is possible, according to the type of authentication used, different restrictions on the rights to be imposed – if a strong authentication is used more rights may be given to the user.

In order to use grid resources, the user has to authenticate first. After a successful authentication the user will be issued a token containing user information, such as user’s roles. For authentication the user contacts the grid portal or a cluster portal and requests authentication. In case the user contacts a cluster portal, the authentication is performed by the Cluster Security Service, which checks the username and password in the Cluster Information Service’s database. If the user contacts the grid portal and the user is not an external user, the portal will send the request to the Grid Security

Service, which in turn will forward it to the Cluster Security Service of the cluster where the user account belongs. For this purpose, there will be a small database at the portal for mapping users to clusters. In order such mapping to be possible, every user should have a unique identifier. Digital certificates contain distinguished name, which is globally unique. But at the beginning we will not use certificates, and even when we begin to support certificates for authentication, we may not restrict users to use only this method for authentication. The solution is at registration time a unique identifier for the grid to be issued to every user.

Authorization is the process of determining if the user has the proper rights to perform an operation – e.g., use a service. Every request for a service will be made at the grid portal or some cluster portal. Before making any request, the user should be authenticated. Actually, a direct request for a resource by the user will not be possible. The user will be able to perform two kinds of actions – submit a job for execution, and use a service (since resources will be represented as services). In the first case – job submission – the resources will be reserved by the Resource Management Service (RMS), and not directly by the user. When making the reservation, the RMS will have the user's token so that it will know the user's roles and identity. When the RMS contacts the Information Service (IS) while searching for resources, it will send as a part of the request the user's roles, so that the IS will be able to filter the resources according to the roles. After that, when the RMS decides which resource(s) will use from the list, made by the IS, it will contact the node manager of the resource. The node manager will check if there are any local restrictions for the user's roles and also it may make a request to the Cluster or Grid Security Service to find whether the user really has the roles stated in the token. Thus an additional security check is added to the process of granting access to a resource. It is meant for cases in which a user's token is forged.

In our system there will be no special authorization service. With the chosen security model, authorization is implicitly realized by the Information Service, the resource and service providers when publishing resources and services, and eventually the node manager. Thus the authorization decision mechanism is incorporated in the functioning of the grid system.

Another important issue concerning authorization is the delegation of rights. When a process is started on behalf of the user – for example, a process to do some computation while performing a submitted job – the process should have the rights of the user in order to be able to use resources. That is why at creation time the process will be issued a token with the user's roles. Here it is possible the process's token to include only those user roles, which are needed for its work.

4 GrOSD vs. H2O and MOCCA security models

At this stage of our ongoing research we take as a pilot prototype the H2O-based CCA-compliant MOCCA framework developed by the CoreGRID partners. When comparing the proposed architecture and security model of the GrOSD platform with those found in H2O, we can find several differences and also some common points. The main feature distinguishing H2O from other grid middleware is the separation of

roles of resource owners (providers) and service deployers. This means that the provider may offer only a raw computational resource to share, and the role of service deployment is left to authorized parties (deployers) who are allowed to deploy pluglets into H2O kernels. This is distinct from the standard scenario proposed by OGSA, where services (even if transient) are offered and deployed by resource providers. Such standard scenario may cause a barrier discouraging providers from sharing, especially when the process of installation (deployment) of services is sophisticated and time consuming. H2O sharing model takes much of the burden from resource providers to the deployers, therefore encouraging providers to share, e.g. in P2P metacomputing scenario. We would consider it valuable if GrOSD architecture could also support such a model of resource sharing with dynamic service provision (deployment), as it is in H2O.

The important part of security mechanisms in H2O is involved in the definition and enforcement of security policies. Both resource providers and pluglet deployers may specify their Java security policies, granting detailed set of permissions to the code executed by clients. The policies are based on the JAAS framework and extended with time-based constraints, protecting providers from malicious or erroneous code run by clients as well as restricting access to system resources (filesystem, network, etc.). We believe that these mechanisms, which are implemented in the H2O kernel may be useful for the building of the prototype implementation of the GrOSD platform. We can observe, that the H2O does not implement the role-based security model in the form proposed by GrOSD. The users and their roles in H2O are constrained to the H2O kernel boundary, because of the assumption of independence of kernel providers, who are not assumed to be aware of each other. However, as the H2O is based on the JAAS framework and Pluggable Authentication Modules (PAM), then it should be possible to plug in the authentication method using Cluster Security Service of GrOSD. This possibility and also potential applicability of restricted X.509 proxies as those known from Globus GSI should be subject to more detailed investigation.

Another important observation is that since the focus of the CoreGrid project is on a component approach for programming grid systems, then adopting several features from the H2O to the GrOSD architecture will enable easier integration of the latter with the MOCCA component framework. This will lead to the possibility of running MOCCA component applications on the GrOSD platform, taking advantage of the simplicity and scalability of the lightweight platform for resource sharing, as well as providing a simple and powerful component programming model.

Alternatively, we may consider the possibility of using MOCCA itself as a base component technology for building prototype of GrOSD. Such features as dynamic deployment of components on shared resources using H2O mechanisms, inter-component communication using RMIX and simple programming model should provide a sufficient base for a lightweight, simple and scalable grid platform, adding the modularity and flexibility to the prototype. The detailed elaboration of such possible design will be the subject of our future research agenda.

5 Conclusion

We introduced the security architecture of the GrOSD middleware. Our main purpose in making the design was the simplicity and scalability. We developed a user management model that is not present in this form in the other grid solutions that we have considered. None of the latter gives the possibility to create user accounts both on the local, cluster level, and on the grid level. Moreover, the role based access control security model we chose to use is modified so that the roles are not associated with particular rights, but has descriptive nature and serve two purposes – give access rights and indicate resource usage. Also, the access is specified by every resource and service by listing the roles that may use it. We use this to incorporate access decision logic in the information service by making it filter accessible resources for a particular request. Another feature of our model is the user hierarchy, achieved by the roles with increasing access rights.

The lower level security will be realized by the Java technologies that will be used for the project implementation. We have put these gridlines as basic security model for GrOSD at an early stage of its design. However it is nonetheless important to consider the functionality and technological differences and similarities with other more advanced projects for service supporting gridware such as H2O and MOCCA. As it might be expected the functional similarity leads to the technological compatibility and further to the possibilities for convergence and adoption of security supporting modules between these projects. The process of adoption can be further simplified by the fact that H2O and MOCCA have been developed as a component-based technology.

The next step of our ongoing research will be the consideration of implementing the specified security model by another existing prototype of CoreGRID's WP7 – ProActive, which is a Java library of simple primitives for cluster/grid applications and system tools featuring security in a uniform framework.

References

1. M. Adamski, M. Chmielewski, S. Fonrobert, J. Nabrzyski, T. Nowocien, and T. Ostwald, "Technical Specification for Authorisation Service", Available: http://www.gridlab.org/Resources/Deliverables/D6.2b.pdf, June, 2005
2. AliCE Grid Computing Project, http://www.comp.nus.edu.sg/~teoym/atsuma.htm, June 2005
3. R. Butler, D. Engert, I. Foster, C. Kesselman, S. Tuecke, J. Volmer, and V. Welch.,"A National-Scale Authentication Infrastructure", *IEEE Computer*, vol. 33, No. 12, , pp. 60-66, 2000
4. DataGrid Security Design, DataGrid Security Co-ordination Group, EU DataGrid project, Available: http://edms.cern.ch/document/344562, June, 2005
5. I. Foster, C. Kesselman, G. Tsudik, S. Tuecke,"A Security Architecture for Computational Grids," in *Proc*. 5th ACM Conference on Computer and Communications Security Conference, 1998, pp. 83-92.
6. Globus Toolkit, http://www.globus.org/toolkit, May, 2005
7. GridLab project, www.gridlab.org, June, 2005

8. GridLab Security Architecture, Available: http://www.gridlab.org/WorkPackages/wp-6/index.html, June, 2005
9. H2O Project, www.maths.emory.edu/dcl/h2o/, July, 2005
10. JGrid project, http://pds.irt.vein.hu/jgrid_index.html, April, 2005
11. Z. Juhasz, K. Kuntner, M. Magyarodi, G. Major, and S. Pota, JGrid Requirements Document, Department of Informaiton Systems, University of Veszprem, Available: http://pds.irt.vein.hu/jgrid/documentation/JGrid_Requirements.pdf, May, 2005
12. M. Kosiedowski and P. Slowikowski, "Authentication and access control in portals: the PROGRESS grid access environment," Presented at Polski Internet Optyczny: Technologie, Uslugi i Aplikacje – PIONIER 2003 conference, April, 9th-11th 2003, Poznan, Poland, Available: http://progress.pscn.pl/English/auth_progress_pioneer2003.pdf, July,2005
13. D. Kurzyniec, T. Wrzosek, D. Drzewiecki, and V. Sunderam. "Towards self-organizing distributed computing frameworks: The H2O approach,"*Parallel Processing Letters*, vol. 13, No. 2, pp.273–290, 2003.
14. M. Magyarodi, Department of Informaiton Systems, University of Veszprem, "The Security Architecture of the Jgrid System", Available: http://pds.irt.vein.hu/jgrid/documentation/JGrid_-security.pdf, May, 2005
15. M. Malawski, , D. Kurzyniec, and V. Sunderam, "MOCCA - Towards a Distributed CCA Framework for Metacomputing," Presented at the 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2005), 2005, Available: http://mathcs.emory.edu/dcl/h20/papers/h2o_hips05.pdf ,August, 2005
16. L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke,"A Community Authorization Service for Group Collaboration," *in Proc.* IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, 2002, p.50.
17. Permis project, Available: http://www.permis.org, June, 2005
18. PROGRESS Portal, Availabe: http://progress.psnc.pl, June, 2005
19. Resource Access Decision, Version 1.0. Available: http://www.omg.org/technology/documents/formal/resource_access_decision.htm, June, 2005
20. Y. M. Teo and X.B. Wang, "ALiCE: A Scalable Runtime Infrastructure for High Performance Grid Computing," *in Procs.* IFIP International Conference on Network and Parallel Computing, 2004, pp. 101-109

Performance monitoring of GRID superscalar applications with OCM-G*

Rosa M. Badia[3], Marian Bubak[1,2], Włodzimierz Funika[1], Marcin Smętek[1]

[1] Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland
[3] Universitat Politecnica de Catalunya, Spain
rosab@ac.upc.edu,{funika,smetek,bubak}@uci.agh.edu.pl
phone: (+48 12) 617 44 66, *fax:* (+48 12) 633 80 54

Abstract. In this paper, the use of a Grid-enabled system for performance monitoring of GRID superscalar-compliant applications is addressed. Performance monitoring is built on top of the OCM-G monitoring system developed with the EU IST CrossGrid project. The design concept of the OCM-G allows for easy adaptation to the monitoring of GRID superscalar applications. We discuss the issues related to performance analysis of GRID superscalar applications as well as those related to the architecture and implementation. At the end a case study of performance monitoring is presented.

Keywords: grid computing, performance analysis, monitoring tools, GRID superscalar, OCM-G

1 Introduction

An important role in any distributed system and especially in Grid environments is played by performance monitoring tools. This is due to the fact that performance and monitoring information is required not only by the user to get information about the infrastructure and the running applications, but also by most Grid facilities to enable correct resource allocation and job submission, data access optimization services, and scheduling. The complexity and dynamics of Grid environments makes that various entities including infrastructure elements, applications, middleware, and others, need to be monitored and analyzed in order to understand and explain their performance behavior on the Grid.

The GRID superscalar (GS) [2], one of approaches to Grid computing, supports the development of applications, in a way transparent and convenient for the user. Its aim is to reduce the development complexity of Grid applications to the minimum, in such a way that writing an application for a computational Grid can be as easy as a sequential program. The idea assumes that a lot of applications is based on some repeating actions, e.g. in form of loops. The granularity of these actions is of the level of simulations or programs, and the data

* This research is partly funded by the EU IST CoreGrid project.

objects will be files. The requirements to run the sequential-fashion application on a Grid are expressed as a specification of the interface of the tasks to be run on the Grid and calls to GS interface functions and link with the run-time library.

GS provides an underlying run-time environment capable of detecting the inherent parallelism of the sequential application and performs concurrent task submission. In addition to a data-dependence analysis based on these input/output task parameters which are files, techniques such as file renaming and file locality are applied to increase the application performance. The run-time is underlied by the Globus Toolkit 2.x APIs [3].

The above reasons motivated a design of a monitoring facility that supports development of applications to be run in the Grid environment using the GS system, to get deeper insight into how an application behaves in such an environment, to help in its effective and fault-tolerant execution. Unfortunately, existing monitoring systems which provide off-line access to monitoring data do not allow to analyse and react on-line to the performance problems arising within the application's execution.

In this paper, we focus on a concept and some implementation ideas of adapting the OCM-G system to support GS applications. Its role is to help the user or an automatic facility to decide on when a performance problem is encountered.

For performance monitoring we use the Grid-enabled OMIS Compliant Monitor (OCM-G) [7] which is an application monitoring system developed in the CrossGrid project [4]. Its features (described in details in Section 3) allow to fit it well into the requirements of running an application on the Grid. In particular, we discuss what metrics are important to assess the performance of the application, these related to standard metrics like an operation time as well application specific metrics, expressed in a special language PMSL allowing the user to define performance indicators most meaningfully giving the context-dependent features of the application and how to get them. Then we come to the general architecture of the functioning of OCM-G in the GS, and its implementation details.

This paper is organized as follows: Section 2 outlines the work related to monitoring GS applications. Section 3 provides an overview of the OCM-G monitoring system. In Section 4 we describe adapting OCM-G to the constraints of GS applications and some implementation issues. In Section 5 we show a case study of using the monitoring system for an example application. Section 6 sums up the results and shows plans for the further research.

2 Related work and requirements to GS monitoring

The ability to monitor and control the elements of a GRID superscalar enabled application is useful for its efficient execution and the environment itself. Nowadays, there are a large number of monitoring tools that address various aspects of Grid computing. Some of them are dedicated for Grid environments, while others were originally developed for distributed computing. Most of tools are mainly designed for infrastructure monitoring. Paraver [5] is a tool, which comes

from distributed/parallel computing, provides performance information on GS applications with a lot of informative displays. Its main drawback lies in an off-line-oriented mode of operation, so it does not allow to undertake actions whenever interesting events occur.

There are a number of systems for monitor Grid infrastructure like Ganglia[9] or JIMS [8], however there are not designed for application monitoring. R-GMA[10] follows the semi on-line monitoring approach using a concept of monitoring data storage.

To provide meaningful information on the performance of GS applications, a monitoring system needs to supply monitoring data in on-line mode, preferably to operate in the event-action paradigm which allows to react properly whenever an interesting event occurs. The monitoring system should function as a distributed system to avoid problems related to a centralised system. Moreover, it should enable to provide performance data in such a way so to make it as application-specific as possible. This could be achieved by introducing a high level performance specification language.

Due to these requirements, we have decided to use the OCM-G [7] described below, since it is Grid-enabled and compliant with monitoring standards [1]. The modular architecture of the OCM-G separates the actual monitoring system from the tools that gather and analyze selected monitoring data. This feature fosters mutual independence of system components and enables users to use their own tools to monitor application performance without any additional effort.

To assess the performance features of an application running in GS environment, we need to analyse such metrics as *Communication volume* and associated overhead, *Overhead due to task synchronization*, *Time of data dependency solving*, *File forwarding time*, *Task submission time*, *Task execution time*, *Resource availability time*. Part of this data can be based on getting relevant events captured by the monitoring system. Otherwise performance evaluation would need accessing data from the GS run-time.

3 Grid-enabled OMIS-compliant Monitoring

The Grid-enabled OMIS-compliant Monitoring system (OCM-G) comprises an infrastructure which enables runtime monitoring of Grid applications. The OCM-G is an autonomous, distributed, decentralized system which exposes monitoring services via a standardized interface called OMIS [1].

Per-host Local Monitors and per-site Service Managers (SM) constitute the distributed part of the OCM-G. Main Service Manager (MSM) distributes data to and collects it from per-site SMs. The Main Service Manager exposes the functionality of the system to performance analysis tools. Owing to the standardized protocol used between the monitoring system and possible tools, OCM-G can be easily adapted to the architecture described in Section 4.

The most important features of the OCM-G include [7]:

- *Transparency* of service-oriented operation is resulting from the fact that the user does not need to manually instrument the application (except the

special case of user-defined events, when there is no other way to do this); instead, pre-instrumented libraries are provided.
- *Flexibility* implies that OCM-G does not limit metrics to a fixed semantics. Instead, a combination of several services is used to obtain a specific metric. This allows the user to derive metrics with the desired semantics and it also enables user-defined metrics.
- *Low monitoring overhead* is achieved via selective runtime instrumentation which can be activated and deactivated at runtime achieved; monitoring data is locally buffered and preprocessed in a distributed way.

The OCM-G has been successfully used with an independent performance analysis tool for Grid applications, the G-PM [6]. G-PM was developed to provide standard metrics for Grid applications (data volume/timing/number of times related to various aspects, like communication, synchronization, resource usage) as well as high-level user-defined metrics meaningful in the context of the application, which can be based on standard metrics and probes and expressed in the PMSL language.

4 Monitoring GS–based Applications: Concept and Implementation Issues

In the present paper we propose an adaptation of the ideas underlying the solutions for the monitoring of Grid applications with OCM-G [7] to different constraints defined by GS–based application features. Below, we describe the architecture and implementation of a monitoring system for GS applications.

The architecture of the GS application monitoring system is presented in Fig. 1. Between GS application and GS run-time, we insert an additional event triggering wrapper. The wrapper transparently passes calls from an application to GS run-time, and sends all required monitored data (name of a GS method, and call's parameters) to the OCM-G.

Our system also supports monitoring other user events, e.g. it can send all kinds of messages (integers or floating numbers, character strings) to the monitoring application. This is done using *probes*, functions that trigger events. A tool which is connected to OCM-G can enable/disable probes, to activate/deactivate the event triggering.

By using OCM-G, it is possible to gather all information needed by a tool interfacing the user or by an automatic tool. In particular, the system can monitor invocations of GS primitives: `GS_On()`, `GS_Off()`, `Execute()`, `GS_Open()`, `GS_Close()`, `GS_Barrier()`, etc. The system allows for gathering the data needed for performance metrics such as the amount of data sent, time of data transmission and process identifiers. The OCM-G architecture allows to control the amount and frequency of monitored data sent from the monitored application to the monitoring system. To avoid unnecessary traffic and overhead, data is sent to the monitor only if required by a consumer.

To allow the system under discussion to monitor GS–based applications, some specific start-up steps have to be performed:

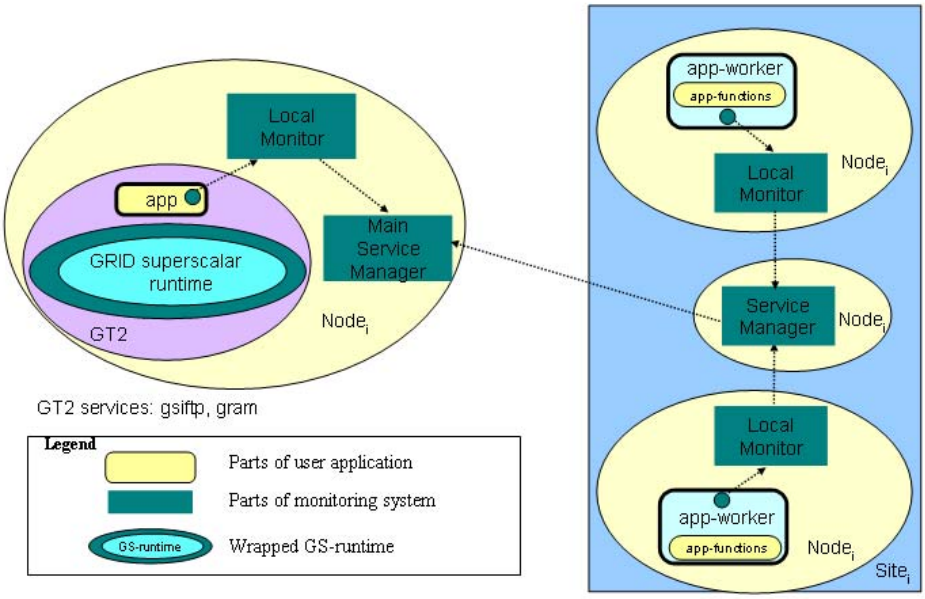


Fig. 1. Monitoring of GS–based applications - architecture

- OCM-G should be initialized from within the application code,
- *OCM-G Main Service Manager* should be up and running,
- An application should be started (this step should be performed with additional parameters pointing to the *Main Service Manager*. During this step, the *Service Manager (SM)* and the *Local Monitor(LM)* are created automatically, if needed, by the application process, or the process can connect to SMs and LMs that have already been created beforehand,
- Now, any component that can communicate using the OMIS [1] interface can connect to the *Main Service Manager*, subscribe to the required probes and receive monitored events of interest, to make decisions e.g. about moving the application to another Grid node if necessary.

To enable intercepting calls to GS run-time primitives we instrumented the GS run-time library. OCM-G distribution provides a tool that performs instrumentation of application libraries in order to trigger events related to enter/exit to library functions. A developer must provide a library and so-called instrumentation description file that specifies which functions inside the library should be instrumented. The description file also specifies for each instrumented function, which of its parameters should be sent to the monitor when a function is called. We also take into consideration adapting the existing instrumentation tool for dynamic libraries, what would be a valuable addition to OCM-G distribution.

Distributed and parallel applications usually consist of concurrently executed processes. In order to identify which processes belong to the same application OCM-G introduces the application name as an `id`. Each process in order to be registered in OCM-G monitoring system must be run with the application

name and with the address of Main Service Manager. Usually it can be done by passing all this information to an application command line. The GS also uses the application name to bind distributed parts of the application. A part that resides on client machine and acts as a front-end of the application for the user is called *master*, and parts that are executed on the computational GRID are called *workers*. Both OCM-G and GS use similar models to deal with the distributed application, what simplifies the integration of these systems. GS supports application deployment process with a set of tools provided with GS distribution that make it almost automatically. In order to allow the monitoring of an application execution, in some stages of this process some modifications must be carried out. One code line that is responsible for a process registration into OCM-G must be inserted to the master as well to the worker programs. Next, both programs must be compiled using OCM-G wrapper script that links with additional OCM-G related objects and libraries. We illustrate these activities within a case study.

An application prepared in this way can be controlled and observed using OCM-G. The inserted instrumentation code sends all required monitoring data to the monitor. All these operations are completely transparent to the user application and are performed with no changes to the original GS runtime source code which is not freely available.

5 Case Study

Within our implementation efforts, we made use of a example application delivered within GS binary distribution, performing parallel multiplication of matrices. We tested it on Intel Pentium 4 1.7 MHz platform. The parallel algorithm carries out matrix multiplication by blocks. To perform this work, the program must do eight additions and multiplications. Each block multiplication and addition is an independent task executed by a worker, so Grid superscalar must spawn eight workers. All computations involve two stages (each executed by four workers) due to dependencies that result from the chosen algorithm. The application is started up simply by executing the master process with the provided information about the application name and address of Main Service Manager:

```
./matmul --ocmg-appname matmul --ocmg-mainsm 959c6326:81f4
```

The master process registers itself into the monitoring system by creating a Local Monitor (unless it exists), which connects to Main Service Manager. At the beginning the master is suspended and can be resumed from any OCM-G compliant tool which connects to Main Service Manager. If a tool makes the master to continue, GS spawns first four workers that register themselves into the monitoring system in the same way as the master did. It is done by changing the `workerGS.sh` script, which resides in a worker's directory. This script is used to start a new worker with GS specific parameters. The following lines illustrate a part of this script, prepared for monitoring:

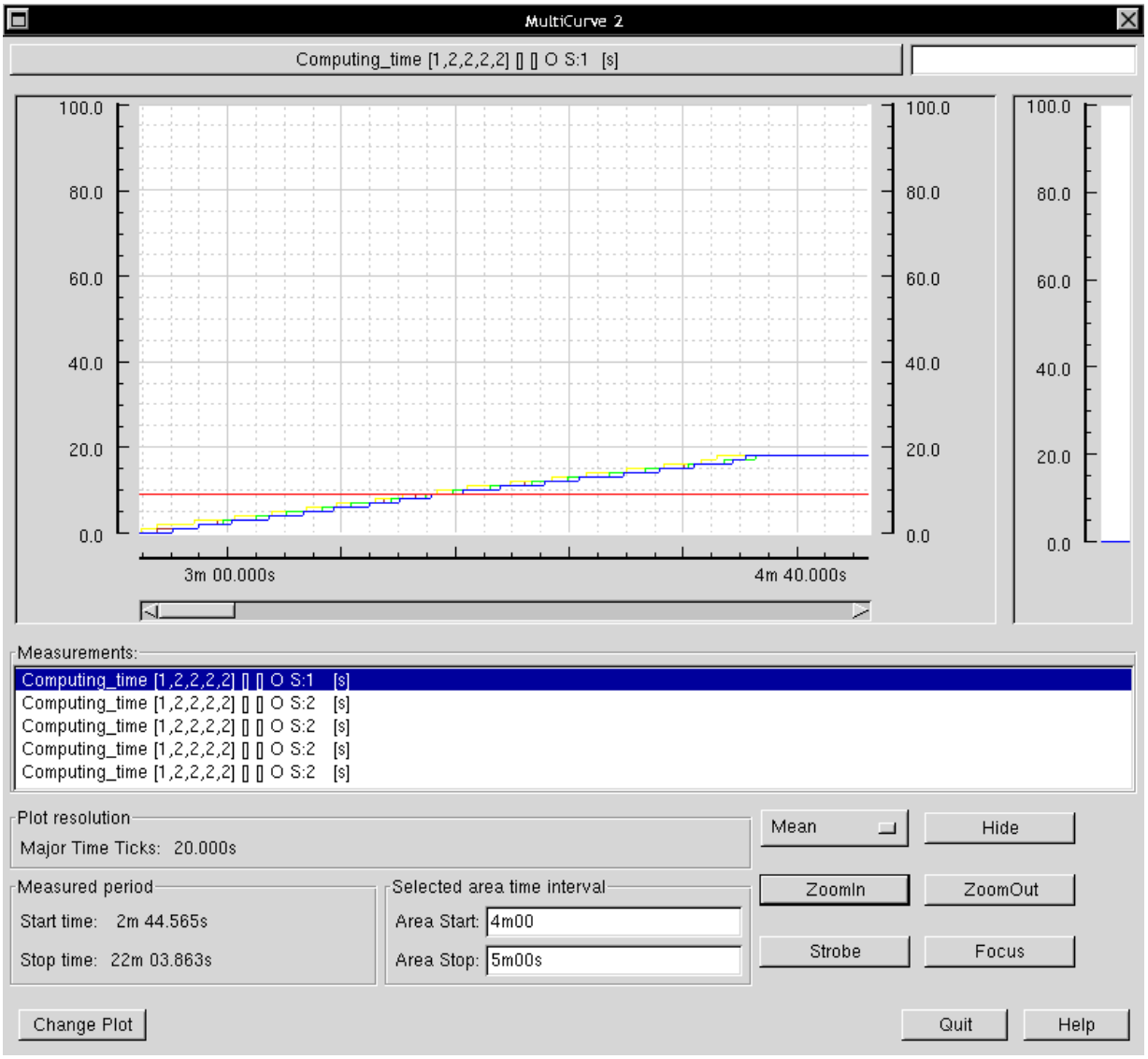


Fig. 2. Example monitoring session using the G-PM performance measurement tool

```
exec ../matmul-worker "$@" --ocmg-appname matmul
     --ocmg-mainsm 959c6326:81f4
```

As we can see the OCM-G related parameters are the same for the master and workers. Starting with this moment the master is resumed (it is waiting for the completion of all workers) and four workers are suspended. We used the G-PM tool to perform an example monitoring session of the created application. First, we defined a measurement (*computing time*) then we chosen a display (multicurve) with an update interval 300 ms and started the application that resumed all suspended workers. The resulting measurements are shown in form of screenshot in Figure 2. The curves present aggregated values of computing time of particular workers (increasing curves) and the master (horizontal line). The observation shows that all workers are consuming CPU time more or less coherently, so in principle it is an example of well performing application. A drawback of G-PM in case of a GS-compliant application is that it is not able to monitor workers which are spawned later (the second stage of this application), at

the moment the G-PM is able to monitor the processes that are already running before its start-up. This drawback will be subject of our further research.

6 Summary

Distributed GS–based applications usually require access to large-scale computing resources. This fact poses a need for a system that handles the execution of such applications and ensures their effective performance and robustness.

In this paper, we focused on an important part of the system, a Grid-enabled monitoring system for GS applications. It is based on the OCM-G monitoring system [7]. We concentrated on the design, implementation issues, and preliminary results of this work. The achieved output is used in the G-PM performance evaluation tool to evaluate when the performance of the application should be improved. Further research will address improving the functionality of OCM-G amd G-PM, building other GS-related metrics, and adapting PMSL to GS applications to enable defining high level metrics meaningful in the context of GS applications.

References

1. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, (1997) `http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`
2. Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Peréz, José M. Cela, and Rogeli Grima: Programming Grid Applications with GRID Superscalar, Journal of Grid Computing, vol. 1, 2003, pp. 151-170.
3. The Globus Project homepage: http://www.globus.org
4. http://www.eu-crossgrid.org
5. http://www.cepba.upc.es/paraver
6. R. Wismueller, M. Bubak, W. Funika, B. Balis, A Performance Analysis Tool for Interactive Applications on the Grid, Intl. Journal of High Performance Computing Applications, vol. 18, no. 3, pp. 305-316, 2004.
7. B. Baliś, M. Bubak, W. Funika, T. Szepieniec, R. Wismüller, and M. Radecki. Monitoring Grid Applications with Grid-enabled OMIS Monitor. In F. Riviera, M. Bubak, A. Tato, and R. Doallo, editors, *Proc. First European Across Grids Conference*, pages 230–239. Springer, Feb. 2003.
8. K. Bałos, L. Bizoń, M. Rozenau, and K. Zieliński. Interoperability Architecture for Grid Monitoring Systems. In M. Bubak, M. Noga, and M. Turała, editors, *Proceedings of Cracow Grid Workshop CGW'03*, Kraków, 2003.
9. Ganglia - monitoring and execution environment http://ganglia.sourceforge.net/
10. http://www.r-gma.org

Towards Semantics-Based Resource Discovery for the Grid

William Groleau[1*], Vladimir Vlassov[2], Konstantin Popov[3]

[1] INSA, Lyon, France. http://www.insa-lyon.fr/
[2] KTH/IMIT, Kista, Sweden. http://www.imit.kth.se
[3] SICS, Kista, Sweden. http://www.sics.se

Abstract. We present our experience and evaluation of some of the state-of-the-art software tools and algorithms available for building a system for Grid service provision and discovery using agents, ontologies and semantic markups. We conducted this research because we believe that semantic information will be used in every large-scale Grid resource discovery, and the Grid should capitalize on existing research and development in the area. We built a prototype of an agent-based system for resource provision and selection that allows locating services that semantically match the client requirements. Services are described using the Web service ontology (OWL-S). We present our prototype built on the JADE agent framework and an off-the-shelf OWL-S toolkit. We also present preliminary evaluation results, which already suggest that representation of semantics information and in particular existing solutions for reasoning on the semantic information need major improvements.

1 Introduction

The Grid is envisioned as an open, ubiquitous infrastructure that allows treating all kinds of computer-related services in a standard, uniform way. Grid services can be described, located, purchased or leased, used, shared. For specific needs services can be composed to form new services. The Grid is to be become large, decentralized and heterogeneous. These properties of the Grid imply that service location, composition and inter-service communication needs to be sufficiently flexible since services being composed are generally developed independently of each other [20], [19], and probably do not match perfectly. This problem should be addressed by using semantic, self-explanatory information for Grid service description and inter-service communication [19], which follows and capitalizes on the research and development in the fields of multi-agent systems and, more recently, web services [17].

We believe that basic ontology- and semantic information handling will be an important part of every Grid resource discovery, and eventually – service composition service [18], [21], [22]. W3C contributes the basic standards and tools, in particular the Resource Description Framework (RDF), Web Ontology Language (OWL) and Web service ontology (OWL-S) [16]. RDF is a data model for entities and relations

[*] The work was done when the author was with the KTH, Stockholm, Sweden.

between them. It provides a simple semantics for this model and a representation schema in XML syntax. OWL extends RDF and can be used to explicitly represent the meaning of entities in vocabularies and the relationships between those entities. OWL-S defines a standard ontology for description of Web services. Because of the close relationship between web- and Grid services, and in particular - the proposed convergence of these technologies in the more recent Web Service Resource Framework (WSRF), RDF, OWL and OWL-S serve as the starting point for the "Semantic Grid" research.

In this paper we present our practical experience and evaluation of the state-of-the-art semantic-web tools and algorithms. We built an agent-based resource provision and selection system that allows locating available services that semantically match the client requirements. Services are described using the Web service ontology (OWL-S), and the system matches descriptions of existing services with service descriptions provided by clients. We extend our previous work [2] by deploying semantic reasoning on service descriptions. We attempted to implement and evaluate matching of both descriptions of services from the functional point of view (service "profiles" in the OWL-S terminology), and descriptions of service structure (service "models"), but due to technical reasons succeeded so far only with the first.

The remainder of the paper is structured as follows. Section 2 presents some background information about semantic description of Grid services and matchmaking of services. The architecture of the agent-based system for Grid service provision and selection is presented in Section 3. Section 4 describes implementation of the system prototype, whereas Section 5 discusses evaluation of the prototype. Finally, our conclusions and future work are given in Section 6.

2 Background

2.1 Semantic Description of Grid Services

The Resource Description Framework (RDF) is the foundation for OWL and OWL-S. RDF is a language for representing information about resources (metadata) on the Web. RDF provides a common framework for expressing this information such that it can be exchanged without loss. "Things" in RDF are identified using Web identifiers (URIs) and described in terms of simple properties and property values. RDF provides for encoding binary relations between a subject and an object. Relations are "things" on their own, and can be described accordingly. There is an XML encoding of RDF.

RDF Schema can be used to define the vocabularies for RDF statements. RDF Schema provides the facilities needed to describe application-specific classes and properties, and to indicate how these classes and properties can to be used together. RDF Schema can be seen as a type system for RDF. RDF Schema allows to define class hierarchies, and declare properties that characterize classes. Class properties can be also sub-typed, and restricted with respect to the domain of their subjects and the range of their objects. RDF Schema also contains facilities to describe collections of entities, and to state information of other RDF.

OWL [15] is a semantic markup language used to describe ontologies in terms of classes that represent concepts or/and collection of individuals, individuals (instances of classes), and properties. OWL goes beyond RDF Schema, and provides means to express relationships between classes such as “disjoint”, cardinality constraints, equality, richer typing of properties etc. There are three versions of OWL: “Lite”, “DL” , and “Full”; the first two provide computationally complete reasoning. In this work we need the following OWL :

- *owl:Class* defines a concept in the ontology (e.g. <*owl:Class rdf:ID="Winery"*/>)
  - *rdfs:subClassOf* relates a more specific class to a more general class
  - *rdfs:equivalentClass* defines a class as equivalent to another class

OWL-S [14] defines a standard ontology for Web services. It comprises three main parts: the profile, the model and the grounding. The service profile presents “what the service does” with necessary functional information: input, output, preconditions, and the effect of the service. The service model describes "how the service works”, that is all the processes the service is composed of, how these processes are executed, and under which conditions they are executed. The process model can hence be seen as a tree, where the leaves are the atomic processes, the interior nodes are the composite processes, and the root node is the process that starts execution of the service.

An example definition of an OWL-S service input parameter is shown in Figure 1. In this example, the concept attached to the parameter *InputLanguage* is *SupportedLanguage*, found in the ontology http://www.mindswap.org/2004/owl-s/1.1/BabelFishTranslator.owl. The class of the parameter is *LangInput*, which has been defined as a subclass of *Input* (predefined in the OWL-S ontology) in the namespace *ions*.

```
- <ions:LangInput rdf:ID="InputLanguage">
    <process:parameterType
      rdf:resource=
"http://www.mindswap.org/2004/owl-s/1.1/BabelFishTranslator.owl" />
  </ions:LangInput>
```

Fig. 1. Definition of an OWL-S service parameter

Few basic OWL-S elements need to be considered by matchmakers:

- *profile:Profile* defines the service profile that includes a textual description of the service, references to the model, etc., and a declaration of the parameters:
  - *profile:hasInput* / *profile:hasOutput*
- *process:Input* / *process:Output* defines the parameters previously declared in the profile, and mostly contains the following elements:
  - *process:parameterType* which defines the type of the parameter.

Note that inputs can be defined by *process:input* or *process:output* or by any subclass of *input* or *output*, as in our example Figure 1. Moreover a profile can also be defined by a subclass of *profile:Profile.*

2.2 Matching Services

Matchmaking is a common notion in multi-agent systems. It denotes the process of identifying agents with similar capabilities [3]. Matchmaking for Web Services is

based on the notion of *similar* services [7] since it is unrealistic to expect services to be *exactly* identical. The matchmaking algorithms proposed in [4], [6] and [7] calculate a degree of resemblance between two services.

Services can be matched by either their OWL-S profiles or OWL-S models [1]. In this work we consider only matching service profiles leaving matching of service models to our future work. Matching service profiles can include matching (1) service functionalities and (2) functional attributes. The latter is exemplified by the ATLAS matchmaker [1]. We focus on matching service functionalities as, in our view, it is more important than matching functional attributes. The idea of matching capabilities of services described in OWL-S using the profiles has been approached first in [7] and refined in [4] and [6]. We use the latter extension in our work as it allows more precise matchmaking by taking into account more elements of OWL-S profiles. Other solutions such as the ATLAS matchmaker [1], are more focused in matching functional attributes and do not appear to be as complete as the one we use.

Our profile matchmaker compares inputs and outputs of request and advertisement service descriptions, and includes matching of the *profile types*. A service profile can be defined as an instance of a subclass of the class *Profile*, and included in a concept hierarchy (the OWL-S ServiceCategory element is not used in our prototype). When two parameters are being matched, the relationship between the concepts linked to the parameters is evaluated (sub/super-class, equivalent or disjoint). This relationship is called "concept match".In the example in Figure 1, *SupportedLanguage* would be the concept matched. Next, the relationship existing between the parameter property classes is evaluated (sub/super-property, equivalent, disjoint or unclassified). This relationship is called "property match". In the example in Figure 1, *LangInput* would be the property matched. The final matching score assigned for two parameters is the combination of the scores obtained in the concept and property matches, as shown in Table 1. Finally, the matching algorithm computes aggregated scores for outputs and inputs, as shown below for outputs:

$$\min(\max(scoreMatch(outputAdv, outputreq) \mid outputAdv \in AdvOutputs) \mid outputreq \in reqoutputs)$$

scoreMatch is the combination score of the "concept match" and "property match" results (see Table 1); *AdvOutputs* is the list of all outputs parameters of the provided service; *reqOutputs* is the list of all outputs parameters of the requested service (requested outputs). The algorithm identifies outputs in the provided service that match outputs of the requested service with the maximal score, and finally determines the pair of outputs with the worst maximal score. For instance, the score will be *sub-class* if all outputs of the advertised service perfectly match the requested outputs, except for one output which is a sub-class of its corresponding output in the requested service (if we neglect the "property match" score). The matching algorithm computes a similar aggregated score inputs.

The final comparison score for two services is the weighted sum of outputs-, inputs- and profile matching scores. Typically, outputs are considered most important ([7]) and receive the largest weight. The profile matchmaker returns all matching services sorted by the final score.

When a requestor does not want to disclose to providers too much information about the requested service, the requestor can specify only the service category.

Table 1. Rankings for the matching of two parameters

Rank	Property-match result	Concept-match result
0	Fail Any	Any Fail
1 2 3	Unclassified	Invert Subsumes Subsumes Equivalent
4 5 6	Subproperty	Invert Subsumes Subsumes Equivalent
7 8 9	Equivalent	Invert Subsumes Subsumes Equivalent

3 Architecture

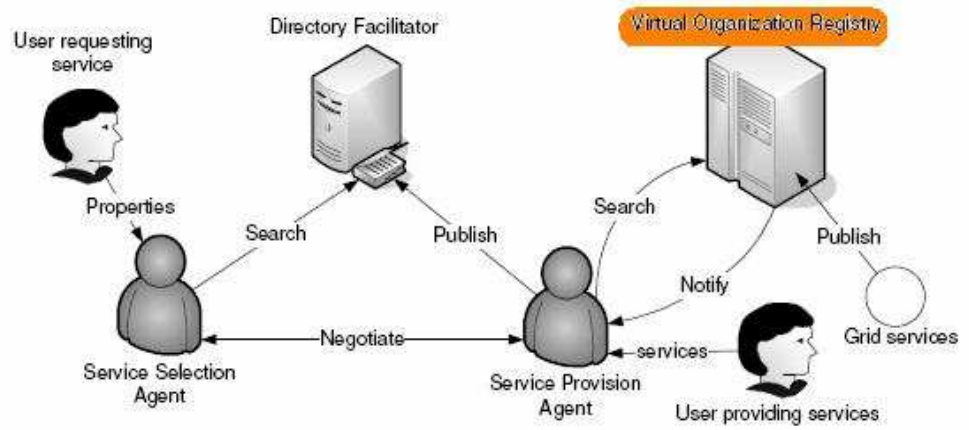


Fig. 2. Architecture of the Agent-Based System for Grid Service Provision and Selection

The architecture of the first system prototype was presented in [2]. In Figure 2, the highlighted "Virtual Organization Registry" becomes obsolete and is replaced by the indexing services of the Globus Toolkit 4 [13]. In the first prototype, a requested service is assumed to be described in GWSDL where properties are defined in Service Data Elements. In the system prototype reported in this article, a requested service is described by the user in an OWL-S document.

The UML sequence diagram in Figure 3 shows how our platform works, and also highlights the matchmaking parts. A service provider specifies URLs of provided services to a Service Provision Agent, which registers (if not yet) to the Directory facilitator. When selecting a service, a user performs the following steps:

0. Instantiation of a Service Selection Agent (SSA);
1. Getting the list of available providers (a.k.a Service Provision Agents, SPA) via the Directory Facilitator (DF);
2. Searching for a matching service, in three steps:
   2.a Sending a description of the requested service as an OWL-S file to the available providers, obtained in Step 1;
   2.b On the provision side, each SPA computes possible matches in parallel;
   2.c The SPAs asynchronously send their results to the requesting SSA;

3. Result treatment, i.e. in our case presenting the matching services to the user.

As we can see, the matchmaking processes occur in the red-marked zone of the diagram, on the provision side. The algorithms implemented at this level are of course either the profile or the model matchmakers.

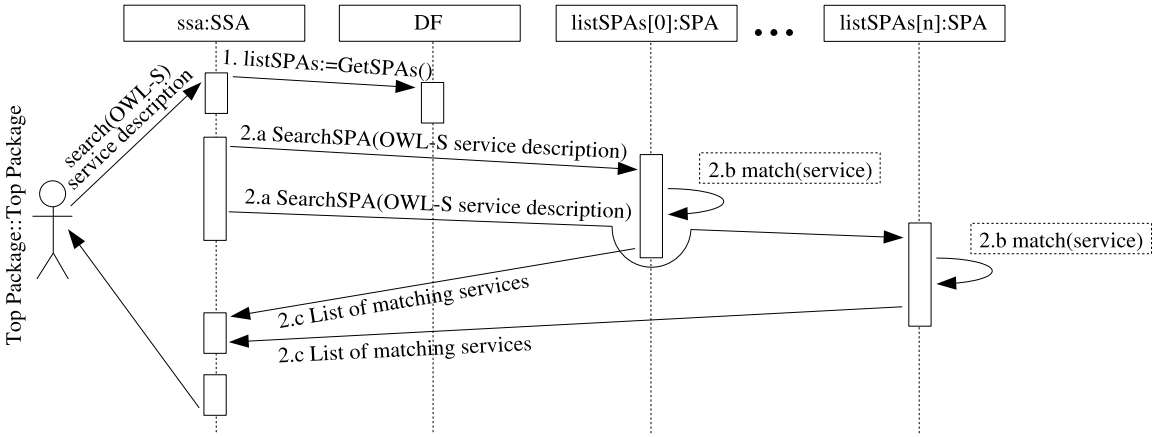


Fig. 3. Selecting services

The use of the category matchmaker (not considered here) is justified in the "secure mode" when a requestor provides only category rather than a detailed description of the service.

The dataflow in the system is depicted in Figure 4. If services are described in GWSDL or in WS-RF, the system should provide WSDL-to-QWL-S or/and WS-RF-to-OWL-S translator like the one used in the first system prototype [2].

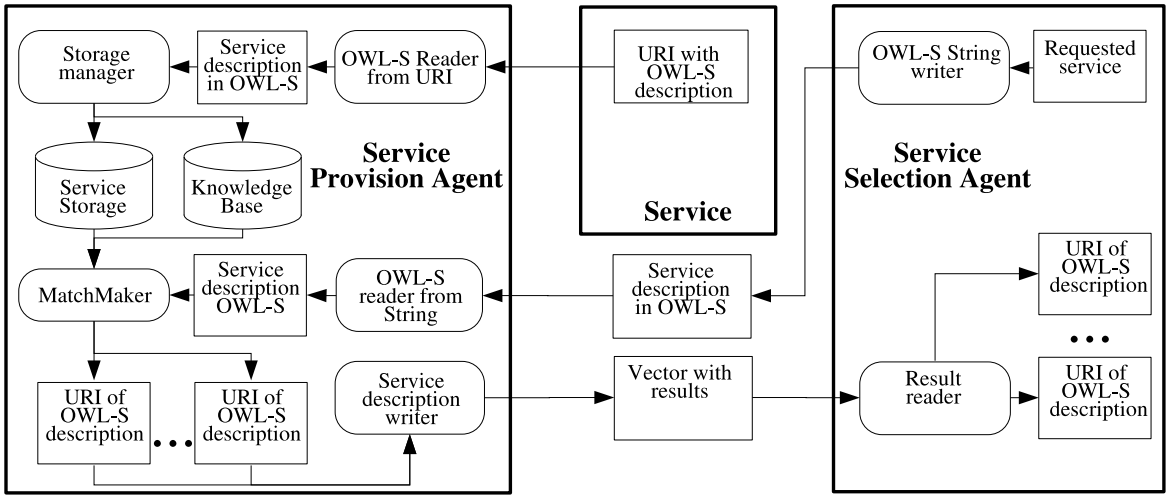


Fig. 4. Information flow in the system

4 Implementation

The first system prototype was reported in [2]. We have upgraded the overall system faithfully to the system specification described in Section 3. We implemented the

profile matchmaker detailed in Section 2. The trickiest part was the implementation of the inference engine where one should be vigilant about limiting the costly calls to the reasoner that confirmed by our evaluation. Ideally, a cache should be provided to remember all computed relationships or matchmaking results, but this has not been implemented and left to our future work. The prototype was implemented using Java 1.4 and the Jade multi-agent platform [9], using the following software and libraries:

- Pellet OWL Reasoner, v. 1.2, [11], which is a free open-source OWL reasoner adapted for basic reasoning purposes and moderate ontologies. We have used Pellet for its good Java compatibility and mostly for its adequacy with our basic needs and for its allegedly good performance with small to moderate ontologies.
- OWL-S API, [8]. This API is one of the available APIs which has no particular advantages (apart supporting the latest OWL-S version). The API has been chosen because it is compatible with the Pellet reasoner.
- Jena v.2.2 [12] – a framework required by the Pellet reasoner.
- Jade [9]. Multi-agent platform on which the system works. We kept Jade which was used in the previous system [2], as this seams to be an efficient platform.
- Jdom v. 1.0 [10]

As mentioned above, the prototype supports only profile matching; we intend to add the matchmaking mechanism for model matching. GUIs have been developed for the providers (letting the possibility to add and remove services) and for the requesters (letting the possibility to search services and modify various search parameters: results collection time, number of providers to contact, specification of the request OWL-S document). A system prototype is available from the authors on request.

5 Evaluation

The implemented prototype has been evaluated using sample services and ontologies found at http://www.mindswap.org and http://www.aktors.org/ontology/portal. In this article, we present only the most significant results of evaluation of the profile matchmaker described in section 2.2. We ran the prototype on a Pentium IV (1.2 GHz). In our evaluation experiments we have considered the following four activities.

- *Determining relationships between classes*. We measure the time spent computing the relationship (sub/super-class, equivalent, disjoint) between two parameters. This computation is performed by the matchmaker when it compares two parameters. In the worst case, this activity takes place three times for the pair of parameters: once for testing the potential equivalence, once for testing the potential subclass relation and once for testing the potential super-class relation.
- *Getting a class in the ontology*. We measure the time spent fetching a class in an ontology, given a URI. As parameters to be matched are given to the matchmaker in the form of URIs pointing to the concepts, they need to be retrieved from the locations and converted into the internal representation used by the reasoner to infer relationships. This activity takes place each time the matchmaker needs to infer a relationship between two parameters.

- *Parsing services*. We also measure the time spent in parsing OWL-S documents in order to store the service descriptions in the API internal representation.
- Other activities (excluding communication between agents).

Note that the first two activities are related to matchmaking.

In order to estimate the relative importance of each of these activities, we calculate the total time taken by an activity as a measured time of one invocation multiplied by the number of invocations. For example, 2 classes need to be fetched in the ontology in order to infer one relationship; at worst 6 relationships need to be inferred (3 for the concept match and 3 for the property match) in order to match 2 parameters. Our estimates show that in order to compare a pair of typical services, each with 2 inputs and 2 outputs, in the worst case the following number of activities takes place: "Determining relationships" – 58 times, "fetching a class" – 56 times, and "parsing services" – 2 times. The results obtained are shown in Figure 5. We can see that matchmaking activities in the Pellet reasoner – determining relationships and getting classes – consume in total 97% of the execution time, i.e. 72% and 25%, respectively. Thus, the performance of the system mostly depends on the performance of the matchmaker which, in its turn, mostly depends on the performance of the reasoner.

Therefore, the best indicator of the system performance is the time to infer relationships as a function of the knowledge base size (i.e. the number of concepts in the knowledge base). To estimate this function, we conducted the following experiment. We inserted different ontologies with various numbers of concepts in the knowledge base, and for each ontology we measured the time needed by the reasoner to determine whether two random concepts were linked by a sub-class (subsumption) relationship. Figure 6 shows the results of evaluation experiments.

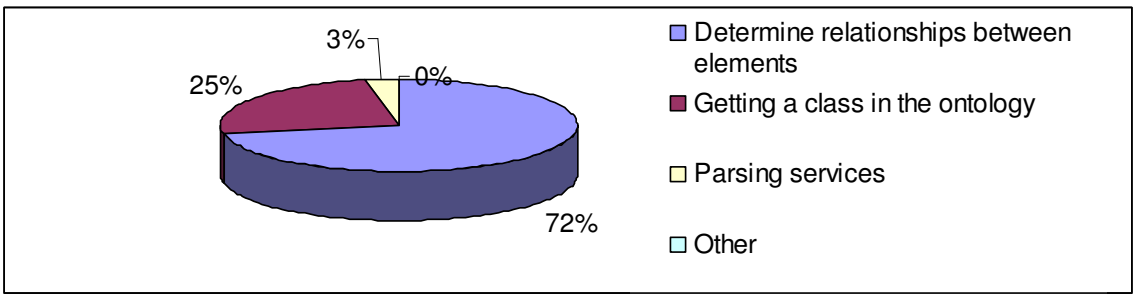


Fig. 5. Time repartition in a matchmaking process

As we can see in Figure 6, the inference time can vary from a few seconds to almost half a minute. Our tests showed that for ontologies with more than 400 concepts (not shown in Fig.6) the inference time suddenly went up to 5 minutes that would make application hardly usable. Thus, the reasoner is a bottleneck in the system, and using a more efficient reasoner would improve the system performance.

We believe that the cause of low performance of the prototype is the high computational complexity of the reasoner algorithm, which is the useful work. There are the following sources of overhead: start-up overhead (the very first call to Pellet involves various loads to memory), and large ontologies (which cause additional overhead when accessing the reasoner's knowledge base).

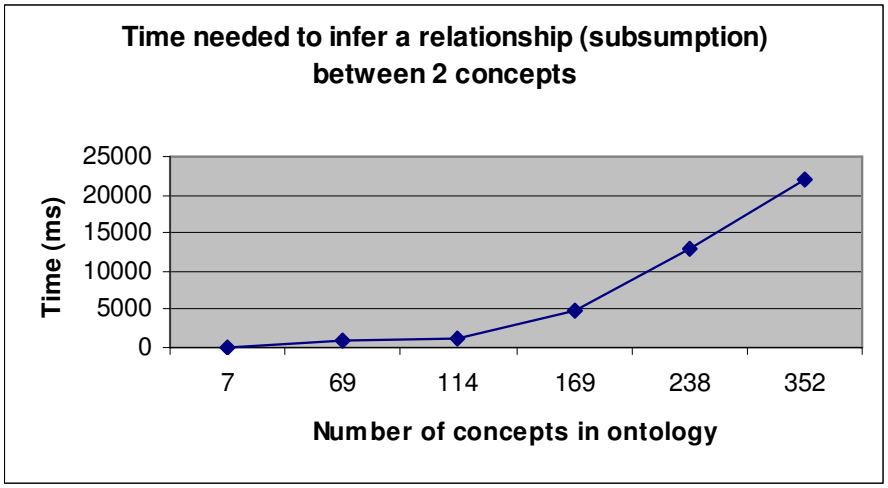


Fig. 6. Relationships inferring time chart

6 Conclusions and Future Work

We presented our experience and evaluation of some of the state-of-the-art semantic-web tools and algorithms. We built an agent-based resource provision and selection system that allows to locate services that semantically match the client requirements. We conducted the research since we believe that basic ontology- and semantic information handling will be an important part of every Grid resource discovery, and eventually – service composition service. In our system prototype we have implemented the matchmaking algorithm proposed in [4]. The algorithm compares a requested service profile with provided service profiles to find a better match(es), if any. Alternatively or complementary, a matching algorithm that compares service models can be used. We intend to consider service model matching in our future work. Our system prototype allows a "secure" mode in which a requester provides only information on service category, and a category matching is done by the providers, whereas profile or model matchmaking is done by the requester.

We have presented an evaluation of the system prototype. We have estimated contribution of different parts of the system to the overall performance. Our evaluation results indicate that the system performance is very sensitive to the performance of the Pellet reasoner used in the prototype which appears to be a bottleneck. We have also shown how the performance depends on the number of concepts in ontology; and the results indicate poor scalability.

Our future work includes improvements of the reasoning performance, research on service composition, and service model matchmaking.

Acknowledgments. This work was supported by Vinnova, Swedish Agency for Innovation Systems (GES3 project 2003-00931). This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265)." The authors would like to acknowledge the anonymous reviewers for their constructive comments and suggestions.

References

[1] Payne, T.R., Paolucci, M., Sycara, K.: Advertising and matching daml-s service descriptions, In Position Papers for SWWS' 01, pp. 76–78, Stanford, USA, July 2001.

[2] Nimar, G., Vlassov, V., Popov, K.: Practical Experience in Building an Agent System for Semantics-Based Provision and Selection of Grid Services – to appear in Proc. PPAM'05, 6-th Int. Conf. on Parallel Processing and Applied Mathematics, Sept 2005.

[3] Klusch, M., Sycara, K.: Brokering and matchmaking for coordination of agent societies: A survey. In Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R., editors: Coordination of Internet Agents: Models, Technologies, and Applications, pp. 197–224, Springer, 2001.

[4] Tang, S.: Matching of Web Service Specifications Using DAML-S Descriptions, Master Thesis, Dept of Telecommunication Systems, Berlin Technical University, March 2004.

[5] Bansal, S., Vidal, J.M.: Matchmaking of Web Services Based on the DAMLS Service Model, In Proc. AAMAS'03, ACM Press, 2003.

[6] Jaeger, M.C., Rojec-Goldmann, G., Liebetruth, C., Geihs, K.: Ranked Matching for Service Descriptions Using OWL-S, KiVS 2005: 91-102

[7] Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic matching of Web-Services capabilities, in Proc of the 1st Int. Semantic Web Conf., pp. 333–347, Springer, 2002.

[8] OWL-S API. http://www.mindswap.org/2004/owl-s/api/

[9] Telecom Italia Lab. Jade 3.1. http://jade.tilab.com/

[10] The JDOM ™ Project. Jdom 1.0. http://jdom.org/

[11] The Pellet OWL Reasoner, http://www.mindswap.org/2003/pellet/index.shtml

[12] HP Labs, Jena http://www.hpl.hp.com/semweb/jena.htm

[13] The Globus Alliance. www.globus.org

[14] http://www.daml.org/services/owl-s/1.1/overview/

[15] http://www.w3.org/TR/owl-features/

[16] World Wide Web Consortium (W3C). www.w3c.org

[17] Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Sci. American, May 2001.

[18] Brooke, J., Fellows, D., Garwood, K., Goble, C.A.: Semantic matching of grid resource descriptions. In Proc. of the 2nd Eur. Across Grids Conference, Nicosia, Cyprus, 2004.

[19] de Roure, D., Jennings, N.R., Shadbolt, N.: The semantic Grid: Past, present and future. Proceedings of the IEEE, 93, 2005.

[20] Foster, I., Jennings, N.R., Kesselman, C.: Brain meets brawn: Why grid and agents need each other. In Proc. of AAMAS'04, New York, USA, July 2004. IEEE.

[21] Heine, F., Hovestadt, M.: Towards ontology-driven P2P Grid resource discovery. In 5th IEEE/ACM International Workshop on Grid Computing, November 2004.

[22] Tangmunarunkit, H., Decker, S., Kesselman, C.: Ontology-based resource matching in the Grid - the Grid meets the semantic web. In ISWC'03, pages 706-721, 2003.

Towards a Scalable and Interoperable Grid Monitoring Infrastructure

Andrea Ceccanti[1], Ondřej Krajíček[2], Aleš Křenek[2],
Luděk Matyska[2], Miroslav Ruda[2]

[1] INFN-CNAF, Bologna, Italy
[2] Institute of Computer Science, Masaryk University Brno, Czech Republic

Abstract. We present an ongoing research in the field of Grid Monitoring Services with the main intent to build an interoperable and scalable monitoring infrastructure which would allow an integration of various existing monitoring sources and frameworks.

1 Introduction

Recently, we proposed Capability-Based Grid Monitoring Architecture (C-GMA) [4, 8] as an extension of the Grid Monitoring Architecture (GMA) [1] concept, to overcome certain GMA shortcomings. However, some of the C-GMA parts are described as logical components only, leaving their internal design to a concrete implementation. The C-GMA is still based on the general concept of producers, consumers and the registry, adding a new service—mediator, responsible for finding matches between producers and consumers.

The main interaction of C-GMA components with the mediator service is advertising their properties and concurrent subscription for notifications on existence of potentially matching parties. This mode of communication matches the publish/subscribe interaction scheme (Sect. 2.2) which is known to promise good scalability even to a very large extent. Therefore, in this paper we propose a distributed implementation of the mediator service based on the publish/subscribe framework.

After a brief review of related work in the following section we introduce the main ideas of the C-GMA in Sect. 3. Sect. 4 presents the core design of the distributed mediator service.

2 Related Work

2.1 Grid Monitoring Architecture

To provide a globally recognised foundation for implementing interoperable tools for Grid monitoring, Global Grid Forum (GGF, `http://www.gridforum.org`) published an informational specification of a basic *Grid Monitoring Architecture* [1], usually abbreviated and denoted as GMA.

GMA provides a very basic view of a monitoring system, which is based on a model consisting of three components: *producer*, *consumer* and *directory service*. The monitoring data are transferred from producer to consumer in the form of events. GMA does not specify any implementation details (such as data presentation mechanisms or communication protocols) but states general implementation requirements (such as scalability of all system components).

Currently, several different Grid monitoring infrastructure implementations exist, e. g. Mercury [10] or R-GMA, Relational Grid Monitoring Architecture [9]. The former (Mercury) describes data types in terms of *metrics*. GMA directory is not present at all—producers and consumers communicate with each other only directly.

The later (R-GMA) is based on the relational data model, using a subset of the SQL language to describe both data and queries. The R-GMA Registry (specific incarnation of the GMA directory service) is replicated in the recent implementation, addressing certain fault-tolerance and performance issues.

2.2 Content-based Publish/Subscribe Systems

Recently, the Publish/Subscribe (P/S) communication paradigm is receiving increasing attention due to its asynchronous, loosely-coupled and flexible style of communication [15]. Applications that leverage this communication paradigm exchange information asynchronously in the form of event notifications produced by *publisher* components that are dispatched to interested *subscriber* components by the P/S middleware. The P/S middleware responsibility is thus to match consumers' subscriptions with published notifications in order to convey messages only where it is explicitly requested.

Content-based P/S (CBPS) systems extend the P/S interaction scheme supporting fine-grained subscription languages that enable subscribers to select very precisely the notification of their interests according to their content. Such systems may be implemented centrally or by a set of distributed brokers that cooperate in the provision of a distributed and coherent communication service. The obvious advantages of a distributed design are the increased scalability, availability, and fault tolerance of the resulting CBPS implementation.

Scalable CBPS systems (e.g., Siena [14]) are thus typically constructed out of a network of brokers that cooperate in forwarding event notifications to remote interested parties. In such distributed design, each broker acts as an access point for the whole CBPS service, collecting subscriptions and dispatching notification for local clients, that may be producers or consumers of information. From an implementation point of view, each broker manages a forwarding table that maps received subscriptions to outgoing interfaces (i.e., network connections towards other brokers or local clients); at forwarding time, notifications are sent only towards local clients or remote destinations that match received subscriptions. This scheme requires that received subscriptions at each broker are broadcasted to all the other brokers in order to consistently establish the routes that are to be followed by published events.

Such routing strategies satisfy two generic requirements: *downstream replication* and *upstream evaluation* [14]. Downstream replication means that a notification should be routed in one copy as far as possible and duplicated only as close as possible along the paths leading to interested subscribers. Upstream evaluation implies that subscription filters are applied on events as close as possible to publishers. The design goal underlying these requirements is to minimise the usage of network resources when routing events to large numbers of distributed subscribers.

Lastly, the principal strength of a CBPS system is that it strongly decouples interacting parties, allowing them to exchange information even if they do not know each other's identity or are not actively participating in the interaction at the same time. Decoupling interacting parties in a Grid environment is crucial due to the highly distributed, dynamic and multi-institutional nature of the resources that can be shared in that environment. For these and other reasons highlighted in [3], we think that the CBPS interaction scheme is suitable for design of the C-GMA distributed mediator service.

3 The C-GMA Architecture and Components

The dark side of the generality of the GMA specification is the fact that it allows multiple implementations that are not mutually interoperable, although being all GMA-compliant.

However, we believe that a diversity of GMA implementations is desirable. It must reflect the diversity of requirements on the implementation of a particular monitoring infrastructure. These requirements may become even contradictory, no single implementation could fulfil them all. For instance, high-grade security vs. high throughput; the security is unavoidable for sensitive information while it imposes limits on throughput due to CPU requirements of encryption/decryption algorithms.

Moreover, different data models may be more suitable for different purposes (cf. R-GMA and Mercury).

For these reasons the goal of the C-GMA is allowing defined *co-existence* and *collaboration* of components coming from diverse GMA implementations rather then proposing a universal architecture. We base our effort on the hypothesis that designing such a universal system is either not possible or it would be too restrictive (e. g. imposing too high overhead).

The component model of the C-GMA is illustrated in Fig. 1. It defines four basic components:

Producer produces the monitoring data in the form of events.

Consumer consumes the monitoring data. Individual consumers are connected directly to the appropriate producers.

Registry (Directory Service) is an information service which stores information about available producers and consumers and also the data type schema.

Mediator is used by consumers and producers to discover potential partners (producers and consumers, resp.). The actual discovery process is described later.

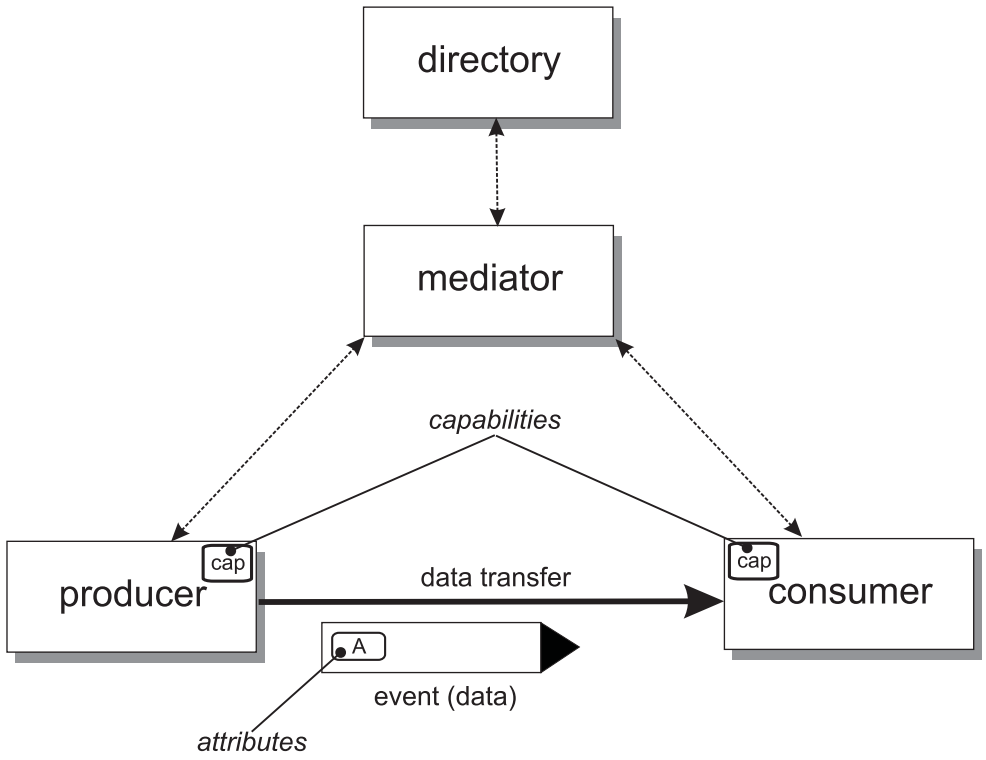


Fig. 1. The C-GMA Component Model

The C-GMA concept works with two *metadata layers*: the *capability and attribute layer* describes properties and requirements of components as well as complementary properties and requirements of data (e. g. a component may be secure or insecure, and data confidential or public). On the other hand, the *data-definition layer* is typically inherited from an existing GMA implementation. The associated metadata describe data types (e. g. table name in the case of R-GMA, metrics for Mercury) of published and requested data, as well as further data specifications (WHERE SQL clause in R-GMA). Matching of producers with consumers is done along two axes then—both data types and capabilities/attributes must be compatible.

Co-existence of several different GMA implementations—"worlds"—could be achieved by defining a particular capability which identifies the "world" the component belongs to. In this way components coming from different worlds (implementations) may co-exist in a single infrastructure without unwanted interference. Moreover, one may instantiate "gates"—components with interfaces to multiple such worlds, capable to convert data from one world (implementation) to another.

A notable extension to the GMA model is the addition of the mediator component. Mediator separates the concepts of producer/consumer discovery and matching from the registration of producer/consumer components. The registration and storage of producer/consumer information is handled by the registry, as in GMA. However, the discovery of producers and consumers is the responsibility of the mediator component.

3.1 Capabilities and Attributes

As outlined above, the requirement on co-existence of different components which are either completely incompatible or specialised for different purposes is addressed in the C-GMA with the additional metadata layer—component capabilities and data attributes.

Components declare via their *capabilities* any features that may affect either the possibility of communication with other components or their ability to handle particular data. Component capabilities are e. g.: *protocol(s)* the component speaks, *ciphers* the component supports, *level of persistence*—once a data event is accepted, it is guaranteed not to be lost e. g. in the case of machine crash, *level of trustworthiness*—which class of sensitive data can be sent to the component.

On the other hand, meta-description of data expressing in which way and to which components the concrete data may be handed over is expressed with data *attributes*. Data attributes may be e. g.: *precious*—this data may not be lost, i. e. should be handled by "persistent" components only; *level of sensitiveness*—results in a requirement on component trustworthiness.

In order to prevent confusion we emphasise again that neither data attributes nor component capabilities are related with event data types. On the contrary, data schema is managed according to GMA, at the C-GMA data-definition layer (see above). Hence capabilities and attributes are properties orthogonal to data types.

The C-GMA assumes a *capability language* which is used to express both: capabilities and attributes. Having the common language for both these entities allows treating them in a symmetric way, namely expressing requirements on capabilities in attributes, and vice versa. The capability language must satisfy certain minimal requirements but no fixed language is prescribed. In particular, the following operations must be supported by the language: *component matching*—given capabilities of a producer and a consumer it must be possible to decide whether these components can communicate with each other, e. g. whether they implement the same protocol; *attribute matching*—given attributes of a piece of data and capabilities of both the producer and the consumer it must be possible to decide whether this producer may handle this data over to this consumer, e. g. whether data security requirements are satisfied.

Currently, we are evaluating two different capability languages. One is XML-based, using XPath expressions to refer from attributes to capabilities and vice versa. The other uses the Classified Advertisements (ClassAds) language [5, 6].

3.2 Mediator—Producer/Consumer Discovery

The mediator is responsible for discovery of appropriate producer or consumer partner. It normally operates actively, by monitoring registrations in the registry and continually evaluating them for potential possible matches between producers and consumers.

As mentioned above, the matching is done along two axes—the components' metadata must match at both capability and data layers. Every time a potential matching pair of component/producer is found, active mediator generates a *proposal* and sends it to both potential parties.

The active mode is complemented with the *passive* mode—mediator can serve requests to discover potential parties based on provided characteristics, i. e. to discover all suitable producers for a particular consumer.

3.3 Component Interaction

In the C-GMA compliant monitoring systems, the following component interactions occur (naming is adopted from Condor Matchmaking [7]):

Advertising – registration of producers and consumers. Besides general information like component identification and address the registration record contains component capabilities and data attributes (if they are uniform for all data) at the capability metadata layer, as well as data description at the data-definition layer. Registration is soft-state, components must renew registration before expiration.

Matching – based on registered metadata, mediator is looking for matching pairs. When new pair is found, both parties are informed about the potential pairing.

Claiming – direct communication between producer and consumer (occurs when a component is notified about the potentially pairing component). Mutual compatibility between components must be verified in this phase by the components.

Starting from this phase, communication occurs only between producer and consumer and it can use a native (not defined by the C-GMA) protocol.

Data transfer – data (events) are send directly between producer and consumer.

The order of component interactions is shown in Fig. 2.

4 The Distributed Mediator

The C-GMA specification, at the time of this writing, defines the mediator as a logically centralised component. The C-GMA specification intentionally does not address implementation of the mediator. To provide a scalable solution for the mediator component, we explore in this section two approaches for distributing the mediator functionality across a network of servers. The obvious objective of

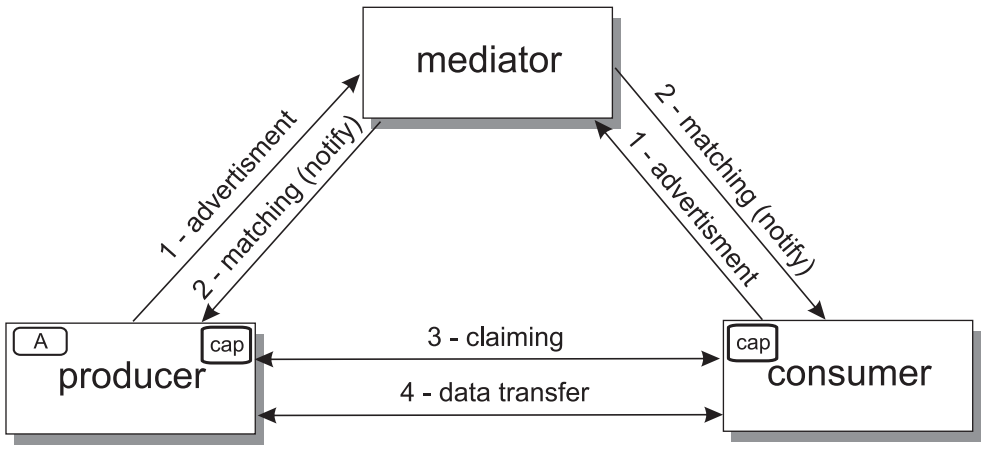


Fig. 2. Interaction of C-GMA Components

our effort is to obtain a reliable, scalable and performant design for the mediator that is free from the drawbacks of a centralised solution.

More specifically, we show how a distributed implementation of the mediator can be designed leveraging the CBPS interaction scheme. The architectural choices we highlight here represents preliminary work and denote a first step towards a real-world distributed mediator prototype implementation.

As introduced in previous sections, in the C-GMA metadata is attached to producers and consumers of information and to data as well. Capabilities describe what components can do, while attributes provide hints to the C-GMA components on how the data itself should be handled. This information together with metadata of the data-definition layer is kept in a document termed the C-GMA *descriptor*. The C-GMA descriptors are used to advertise components' capabilities and attributes and provide the basis for the mediator matchmaking process.

A distributed implementation of the mediator can be architected in several ways. One possibility is to replicate all the descriptors on all the brokers. In this approach (that we tenderly named the *naive* approach), each broker manages the matchmaking for local clients and broadcasts each registered C-GMA descriptor to all the other brokers for further matchmaking. The main advantage of this replication strategy is that it is simple to implement and it provides good fault tolerance (in case of failures, little work has to be done to ensure consistency between the replicas and to redirect orphaned C-GMA components to other active brokers). However, this approach may have significant scalability problems since it considerably wastes network and storage resources by replicating information where it is not needed for the matchmaking process.

Another option is to design the mediator service as an overlay network of distributed brokers that implement a content-based P/S system. In particular, we leverage CBPS so that each mediator broker receives information *only* regarding remote consumer components that are compatible (i.e., whose capabilities and component attributes match) with locally managed producer components.

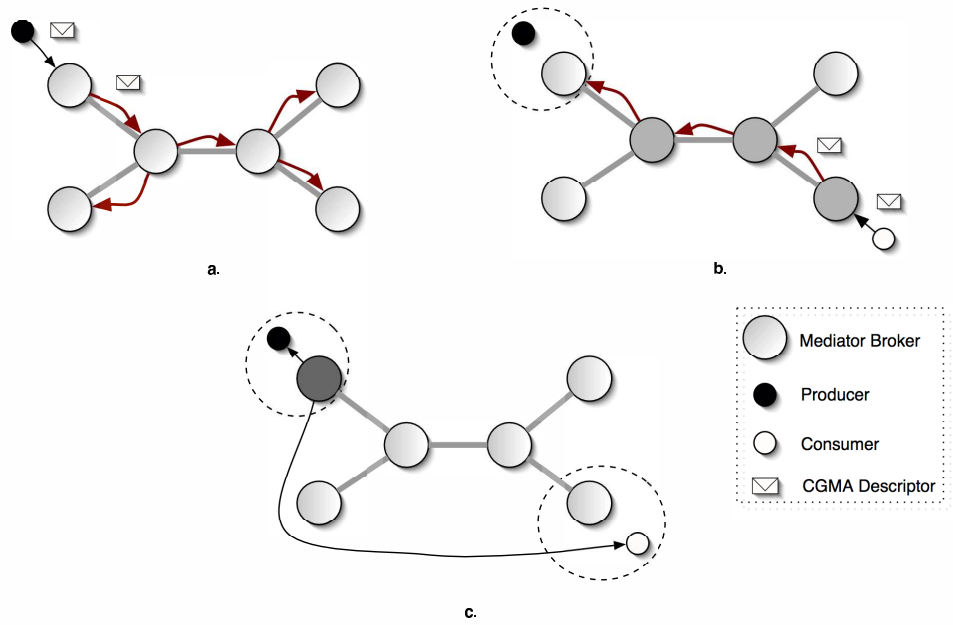


Fig. 3. The distributed mediator content-based replication strategy. Figure a) shows the broadcasting of a producer C-GMA descriptor. In figure b), a C-GMA consumer descriptor is forwarded by each broker towards the matching producer. Finally, in figure c), the last mile broker performs the final matchmaking between components capabilities and data attributes and types and sends a matching proposal to interested C-GMA components.

To do so, we provide each broker with a forwarding table that is built according to registered C-GMA descriptors and we implement a routing strategy that satisfies the CBPS downstream replication and upstream evaluation principles introduced in Sect. 2.2.

In our scheme (see Figure 3), producers drive the interaction. Whenever a producer registers with a mediator broker, two actions are performed: i) locally managed matching consumers are notified of the producer existence and ii) the producer's descriptor is broadcasted to the other brokers. This last step is necessary to ensure forwarding table consistency across all the brokers and correctly establish the routes that C-GMA consumers' descriptors will follow in the CBPS overlay network. More specifically, whenever a mediator broker receives a producer's descriptor from a neighbour, it updates its forwarding table adding the received descriptor to the set of descriptors associated with that specific neighbour.

Consumer C-GMA descriptors are treated differently. Whenever a consumer component registers itself, the local mediator broker starts a matchmaking process comparing its descriptor with:

- locally managed producer descriptors, so that matching producers are immediately notified of the newly arrived consumer;
- producers descriptors appearing in the forwarding table. If a matching descriptor is found, the received consumer descriptor is forwarded towards the matching neighbours for further matchmaking by remote brokers.

The main advantage of the CBPS replication strategy is that it limits the spreading of consumer descriptors only where these are really needed for the matchmaking process. The immediate consequence is a gain in scalability and performance of the infrastructure, since the amount of administrative traffic introduced in the overlay is limited and the distributed matchmaking function is ran only when strictly necessary (i.e., on all the brokers appearing on the shortest path that connects the producer edge broker with the consumer edge broker). In contrast, the naive replication approach states that all C-GMA descriptors are broadcasted to all the brokers so that the matchmaking process is executed on each broker even on descriptors that will not match locally managed C-GMA components.

5 Conclusions

We have described a specific part of an ongoing research aimed at creating scalable and interoperable monitoring architecture for the Grid. The discussed C-GMA architecture offers a general approach to integrate different GMA implementations. The distributed mediator improves the scalability of the C-GMA matchmaking process by leveraging the CBPS communication paradigm. We believe that the resulting architecture could provide highly scalable interoperability framework for various Grid monitoring tools.

Indeed, the proposed CBPS replication strategy is only one of the many approaches that could be conveniently applied to the design of a distributed mediator. We plan to investigate other approaches that leverage recent research results regarding self-organising latency-aware overlay topologies [11], distributed hash tables [12] and epidemic dissemination information protocols [13].

Moreover, the choice of propagating the producer descriptors through the whole network and matching the consumer registrations can be symmetrically replaced by propagating consumer descriptors and matching producer registrations. Assessment of effectivity of these two approaches should be a subject of further evaluation.

Acknowledgement

The work described in this paper is the result of collaboration enabled through the EU Network of Excellence *European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies*, (CoreGRID, FP6-004265), whose support is highly acknowledged. Also, part of this work is also supported by the MU Research Intent MSM0021622419.

References

1. B. Tierney et al., "A Grid Monitoring Architecture", Global Grid Forum Performance Working Group, January 2002.
`http://www.gridforum.org/documents/GFD.7.pdf`

2. Ian Foster, Carl Kesselman, "The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999, ISBN: 1-55860-475-8.
3. A. Ceccanti, F. Panziery, "Content-based Monitoring in Grid Environments", In Proc. of the ETNGrid 2004.
4. J. Sitera et al., "Capability and Attribute Based Grid Monitoring Architecture", In Proc. of Cracow Grid Workshop 2004.
5. R. Raman, "Matchmaking Frameworks for Distributed Resource Management", Dissertation Thesis, University of Wisconsin – Madison, 2001.
6. M. Solomon, "The ClassAd Language Reference Manual, Version 2.4", Computer Sciences Department, Univeristy of Wisconsin – Madison, 2004. `http://www.cs.wisc.edu/condor/classad/refman/`
7. Rajesh Raman, Miron Livny, and Marvin Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing", In Proc. of the Seventh IEEE International Symposium on High Performance Distributed Computing, July 28-31, 1998, Chicago, IL.
8. Křenek, A., et al. C-GMA – Capability-based Grid Monitoring Architecture. CESNET technical report 6/2005. `http://www.cesnet.cz/doc/techzpravy/2005/cgma/`.
9. S. Fisher: Relational Model for Information and Monitoring. Technical Report GWD-Perf-7-1, GGF, 2001.
10. Zoltan Balaton, Peter Kacsuk, Norbert Podhorszki and Ferenc Vajda. From Cluster Monitoring to Grid Monitoring Based on GRM. In proceedings 7th EuroPar2001 Parallel Processings, Manchester, UK. pp. 874-881. 2001
11. A. Ceccanti, G.P. Jesi, "Building latency-aware overlay topologies with QuickPeer", In Proc. of IEEE ICNS 2005.
12. I. Stoica et al., "Chord: a scalable, peer-to-peer lookup protocol for Internet applications", in Proc. of ACM SIGCOMM'01, 2001
13. P.T. Eugster et al., "Lightweight Probabilistic Broadcast", ACM Transactions on Computer Systems, Vo. 21, 2003.
14. Antonio Carzaniga et al., "Design and evaluation of a wide-area event notification service", ACM Transactions on Computer Systems Vol. 19, No. 3, August 2001, pp. 332-383.
15. Patrick Th. Eugster et al., "The many faces of Publish/Subscribe", ACM Computing Surveys, Vol. 35, No. 2, June 2003, pp. 114-131.

Sensor Oriented Grid Monitoring Infrastructures For Adaptive Multi-Criteria Resource Management Strategies

Piotr Domagalski[1], Krzysztof Kurowski[1], Ariel Oleksiak[1], Jarek Nabrzyski[1], Zoltán Balaton[2], Gábor Gombás[2], Péter Kacsuk[2]

[1] Poznań Supercomputing and Networking Center, Noskowskiego 10, 60-688 Poznań, Poland
{domagalski,krzysztof.kurowski,ariel,naber}@man.poznan.pl
http://www.man.poznan.pl

[2] MTA SZTAKI, Budapest, H-1528 P.O.Box 63, Hungary
{balaton, gombasg, kacsuk}@sztaki.hu
http://www.sztaki.hu

Abstract. In a distributed multi-domain environment, where conditions of resources, services as well as applications change dynamically, we need reliable and scalable management capabilities. The quality of management depends on many factors among which distributed measurement and control primitives are particularly important. By exploiting the extensible monitoring infrastructure provided at the middleware level in a grid meta-scheduling service, in particular integration between GRMS (Grid Resource Management System) and Mercury (Grid Monitoring System), it is possible to perform analysis and then make intelligent use of grid resources. These provide the basis to realise dynamic and adaptive resource management strategies, as well as automatic checkpointing, opportunistic migration, rescheduling and policy-driven management, that has attracted attention of many researchers for the last few years. In this paper we present the current status of our ongoing research in this field together with an example of sensor oriented grid monitoring capabilities facilitating efficient remote control of applications and resources.

1 Introduction

Recently developed grid middleware services[1][2][3] allow us to connect together resources (machines, storage devices, etc.) such as computing clusters with local queuing systems to establish a virtual multi-domain grid environment where various calculations and data processing tasks can be performed in a more efficient way. Unfortunately, efficient and flexible remote management of distributed applications and resources is still an issue that must be addressed today. Note, that management is already complex with existing queuing systems and their complexity is expected to reach a new dimension with multi-domain grid environment. Applications submitted to various resources are usually run under the full control of a queuing system running on a gateway node (front-end machine). Internal nodes of these systems are often inac-

cessible from the outside due to private IP addresses (using NAT) or firewalls and queuing systems often provide only basic operations that can be used to control applications remotely. Furthermore, in many grid systems relatively simple, script-based solutions [3] have been adopted to expose capabilities offered by queuing systems what in fact limit the allowed monitoring and control/steering operations that can be performed on jobs running within local clusters to the minimum set of starting/cancelling a job and finding out its status, see figure 1 below.

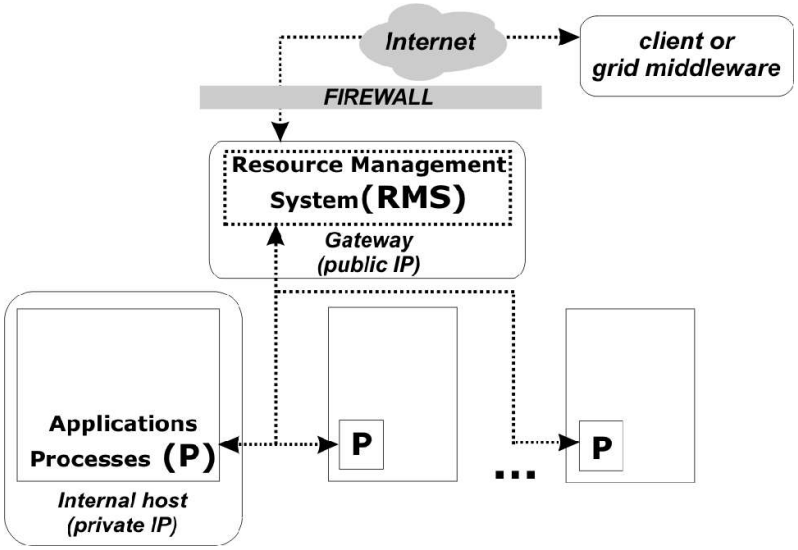


Fig 1. A general architecture of many local resource management systems (i.e. local batch scheduler, local queuing system)

There is also a lack of flexible and adaptive monitoring mechanisms enabling dynamic configuration and reconfiguration of information providers and monitoring tools in case of adding or removing resources. As a step towards better management of grid environments, in this paper we present a set of advanced metrics and sensor oriented features, provided by Mercury monitoring system, which can be exploited by the grid middleware. We believe that new application steering and control routines will help to build more efficient and adaptive resource management strategies suitable for many real scenarios.

2 Motivations

One of the main motivations of our research was to facilitate efficient and dynamic adaptive resource management in distributed environments by a tight integration between grid middleware services, in particular GRMS[4], a meta-scheduling system, and Mercury[5], a grid monitoring system providing reliable distributed measurement of resources, hosts and applications. The second objective was to make use of new monitoring capabilities, in particular embedded non-intrusive application sensors and actuators, and provide a grid middleware with more efficient remote application steering and control mechanisms. We have proposed some extensions to push mechanisms in Mercury enabling clients to configure and dynamically reconfigure certain measurement conditions for applications and resources. In this way, clients or grid middleware

services can be automatically notified when these conditions are met. Finally, we have established a distributed testing environment connecting a few geographically distributed clusters to evaluate the performance of remote application steering and sensor oriented monitoring mechanisms and also to prove the concept of using these capabilities for more efficient and adaptive resource management in distributed environments.
All aforementioned objectives are addressed in this paper within the next sections. In section 3 related works and various distributed monitoring infrastructures are presented. In Section 4 we present example controls and metrics which can be embedded in distributed applications for more efficient remote control. An additional component to Mercury for flexible event or rule based monitoring of applications and resources is discussed in section 5. Example adaptive multi-criteria resource management strategies and potential benefits of using advanced monitoring capabilities are presented in section 6. Finally, section 7 summarizes our research efforts and shows preliminary results.

3 Related works and activities

Monitoring is a very broad term and different grid middleware services and tools are often considered in this category. Specifically grid information systems (e.g. Globus MDS and R-GMA), infrastructure monitoring services (Hawkeye, Ganglia), network (e.g. NWS) and application monitoring tools (e.g. OCM-G) all grouped together under this umbrella, although the functionalities they realise and provide for specific problems are very different. The APART2 project published a white paper [10] containing a directory of existing performance monitoring and evaluation tools with the aim to help grid users, developers and administrators in finding an appropriate tool according to their requirements. Another collection and comparison of several grid monitoring systems can be found in [11].
The Grid Monitoring Architecture (GMA), a recommendation of the Global Grid Forum (GGF), describes the basic characteristics of a grid monitoring system. According to the GMA, data is made available by *producers* and is used by *consumers*. Information discovery is supported by utilizing a *directory service*. Data is transmitted from the producer to the consumer as a series of time-stamped events. The GMA also defines the most important interactions between producers, consumers and the directory service. The GMA however makes no recommendations about the data model or protocols.
In this paper we discuss remote management of distributed applications and resources for which a reliable and efficient monitoring infrastructure is a crucial component which has to support both monitoring and controlling of grid resources as well as applications running on them.
The Mercury Grid Monitoring System is a general purpose grid monitoring system developed by the GridLab project. It has been designed to satisfy requirements of grid performance monitoring: it provides monitoring data represented as metrics and also supports steering by controls. It supports monitoring of different grid entities such as

resources, services and running applications in a generic, extensible and scalable way. Mercury features a modular design with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system. It follows recommendations of the Grid Monitoring Architecture defined by GGF. The input of the monitoring system consists of measurements generated by sensors. A Local Monitor (LM, see figure 2 below) runs on every machine and manages sensors embedded into it or in application processes (P) and also acts as a producer which forwards collected data to a Main Monitor. Note here that many application processes can simultaneously send and receive messages to/from LM . The Main Monitor (MM), which is preferably situated on a gateway, receives requests from a client (consumer) and routes them to Local Monitors, eventually gathering all answers and forwarding them back to the client.

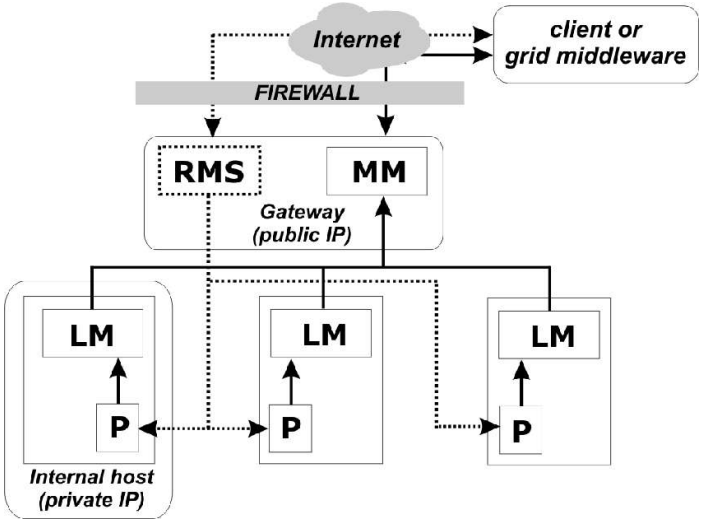


Fig 2. A general Mercury monitoring system architecture

4 Embedding sensors in applications – MPI example

In addition to basic monitoring features, Mercury also allows sensors to be embedded in applications by enabling applications to register their own metrics and controls in order to both publish application specific information and receive steering signals via Mercury. Application specific metrics and controls can be accessed in the same way as any other metric/control. As it is presented in figure 3, a direct two-way communication channel can be established between applications/processes (P) and local monitors (LM) of Mercury thus, external clients, for example a management system, can interact with the application via a main monitor (MM) located on a gateway machine.
Three parameters are associated with every application that is using an embedded sensor to provide a way to uniquely identify a process in a multi-threaded or parallel (for example MPI) application:

- *program name:* this parameter can be used for the human-readable identification of a particular process or thread in a multi-process or multi-threaded application,
- *tid:* this parameter is a machine-readable identification number of processes, e.g. the thread identifier of a multi-threaded application or process' rank in

a MPI application, the program_name tid should be unique for every process of the same job,

- *jobid:* a global job identification usually given by a grid service (such as a scheduler), the local queuing system or operating system (see the RMS component in figure 3).

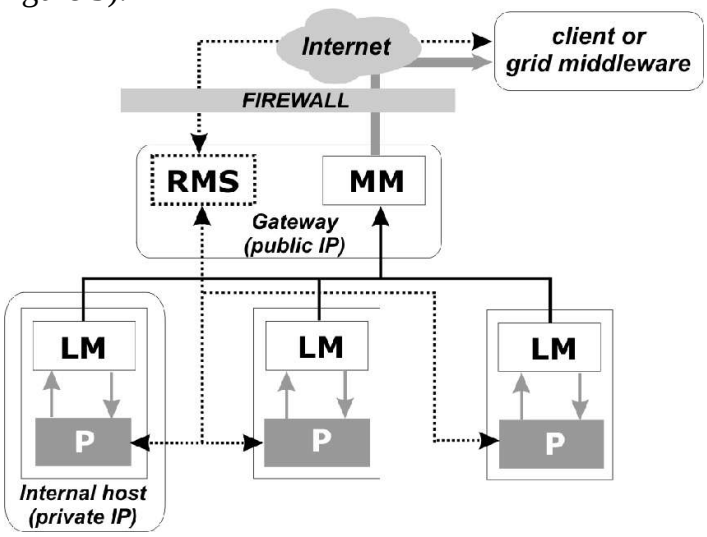


Fig 3. Two-way communication channels between distributed applications and Mercury local monitors (LM).

Once a communication channel is established between P and LM, embedded sensors can be used to interact dynamically with remote applications/processes. Technically speaking applications must be recompiled with non-intrusive and portable Mercury libraries.

Initially, we have implemented and tested the following metrics and controls for MPI applications:

- *progress:* every process in MPI groups works independently; this metric indicates its status, for example as 0-100% value,
- *mpisize:* this metric provides the maximum MPI_Rank identifier, so the external control entity knows how to independently access any of the running processes,
- *whereami:* it is crucial to know which process rank is run on which local host, so this metric provides MPI_Rank and hostname for each process in MPI group,
- *checkpoint:* this control should be interpreted by a master process in a MPI process group. Therefore, it should act properly and inform other processes using MPI to shutdown and write down their status files. This can be used by GRMS to dynamically reschedule the execution of the application and migrate it if necessary.
- *resource usage metrics*: these metrics provide a detailed view of resources (e.g. memory usage, CPU load, free disk quota) consumed by the application.

To access mentioned metrics and controls a client of Mercury, in our case a management system, has to simply query a main monitor (MM) by using a metric name (e.g. *progress*) and appropriate parameters specifying an application identity (*program_name, jobid*). Moreover, the following parameters may also be added in the query:

- *host:* to limit the query to processes on the specified host,
- *tid*: to limit the query to process with the specified MPI process' rank.

In the next section we present more sophisticated monitoring capabilities which are in fact based on above mentioned controls and metrics but provide flexible control mechanisms for a management system.

5 Event and alert monitoring

Mercury provides *push* mechanisms for event-like measurements (e.g. a state change) and basic support for generating periodic events to enable monitoring of continous metrics (e.g. processor load). One of the useful features from the management point of view however, is a generation of events based on processing monitoring data. Since this capability is not strictly part of the monitoring system and in fact needs knowledge about the semantics of the monitored data we have proposed an external module to Mercury called Event Monitor (EM). In a nutshell, EM implements more sophisticated push mechanisms as it is highlighted in figure 4. Event Monitors allow clients dynamic management and control of event-like metrics as very useful information providers for clients or management systems. We see many real scenarios in which an external client wants to have access to metrics described in the previous section (regardless of their type) and additionally, often due to performance reasons, does not want to constantly monitor their values.

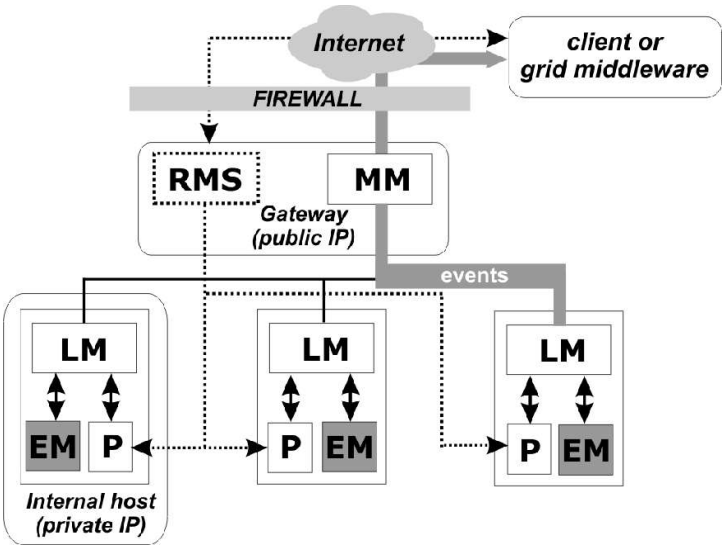


Fig 4. Event Monitors as external mercury modules for event-like monitoring of resources and applications

Nowadays, policy-driven change and configuration management that can dynamically adjust the size, configuration, and allocation of resources are becoming extremely important issues. In many real use cases, a resource management system may want to take an action according to predefined management rules or conditions. For example, when application progress reaches a certain level, the process memory usage becomes too high or dedicated disc quota is exceeded. Event Monitor was developed to facilitate such scenarios. Its main functionality is to allow an external client to register a

metric in Event Monitor for receiving appropriate notifications when certain conditions are met. Strictly speaking, clients can setup an appropriate frequency (a default one has been set as 5 seconds) of Event Monitor requests to LM. They can also use a predefined standard relational operator (greater than, equal to, etc.) and different values of metrics to define various rules and conditions. Example EM rules for fine-grained enforcement of resource usage or application control are presented below:

Example application oriented rules in Event Monitor:
app.priv.jobid.**LOAD**(*program_name, tid*) > 0.8
app.priv.jobid.**MEMORY**(*program_name, tid*) > 100000000
app.priv.jobid.**PROGRESS**(*program_name, tid*) = 0.9

Example host oriented rules in Event Monitor:
host.loadavg5 *> 0.5*
host.mem.free*(host*) < 100 KiB
host.users*(host)* > 0
host.net.total.error*(host, interface*) > 100000

When the condition is fulfilled Event Monitor can generate an event-like message and forward it to interested clients subscribed at the Mercury Main Monitor component - MM. Note that any metric, host or application specific, that returns a numerical value or a data type that can be evaluated to a simple numerical value (e.g. a record or an array) can be monitored this way.
In fact, four basic steps must be taken in order to add or remove a new rule/condition to Event Monitor. First of all, the client must discover a metric in Mercury using its basic features. Then it needs to specify both a relation operator and a value in order to register a rule in Event Monitor. After successfully registering the rule in Event Monitor, a unique identifier (called *event_id*) is assigned to the monitored metric. To start the actual monitoring, the commit control of Event Monitor on the same host has to be executed. Eventually, the client needs to subscribe to listen to the metric (with no IP address of host specified) through Main Monitor and wait for the event with the assigned *event_id* to occur.

6 Example adaptive multi-criteria resource management strategies

The efficient management of jobs before their submission to remote domains often turns out to be very difficult to achieve. It has been proved that more adaptive methods, e.g. rescheduling, which take advantage of a migration mechanism may provide a good way of improving performance [6][7][8]. Depending on the goal that is to be achieved using the rescheduling method, the decision to perform a migration can be made on the basis of a number of events. For example the rescheduling process in the GrADS project consists of two modes: migrate on request (if application performance degradation is unacceptable) and opportunistic migration (if resources were freed by recently completed jobs) [6]. A performance oriented migration framework for the Grid, described in [8], attempts to improve the response times for individual applica-

tions. Another tool that uses adaptive scheduling and execution on Grids is the GridWay framework [7]. In the same work, the migration techniques have been classified into the application-initiated and grid-initiated migration. The former category contains the migration initiated by application performance degradation and the change of application requirements or preferences (self-migration). The grid-initiated migration may be triggered by the discovery of a new, better resource (opportunistic migration), a resource failure (failover migration), or a decision of the administrator or the local resource management system.
Recently, we have demonstrated that checkpointing, migration and rescheduling methods could shorten queue waiting times in the Grid Resource Management System (GRMS) and, consequently, decrease the application response times [9]. We have explored a migration that was performed due to the insufficient amount of free resources required by incoming jobs. Application-level checkpointing has used in order to provide full portability in the heterogeneous Grid environment. In our tests, the amount of free physical memory has been used to determine whether there are enough available resources to submit the pending job. Nevertheless, the algorithm is generic, so we have easily incorporated other measurements and new Mercury monitoring capabilities described in previous two sections. Based on new sensor-oriented features provided by Event Monitor we are planning to develop a set of tailor-made resource management strategies in GRMS to facilitate the management of distributed environments.

7 Preliminary results and future work

We have performed our experiments in a real testbed connecting two clusters over the Internet located in different domains. The first one consists of 4 machines (Linux 2-CPU Xeon 2,6GHz), and second consists of 12 machines (Linux 2-CPU Pentium 2,2 GHz). The average network latency time between these two clusters was about 70ms.

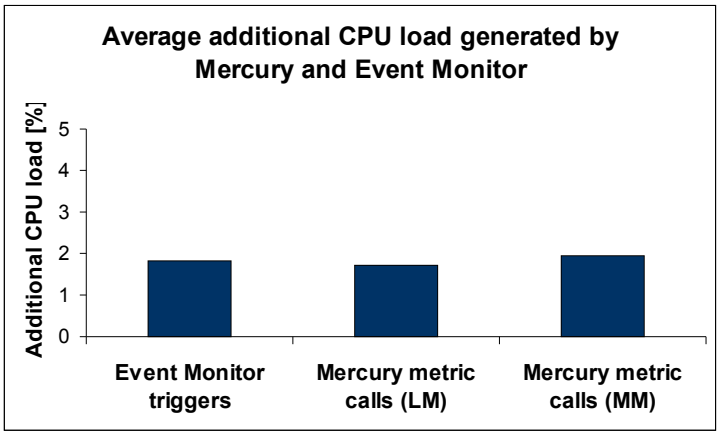


Fig 5. Performance costs of Mercury and Event Monitor

In order to test capabilities as well as performance costs of Mercury and Event Monitors running on testbed machines we have developed a set of example MPI applications and client tools. As it is presented in figure 5 all control, monitoring and event-based routines do not come at any significant performance. Additional CPU load generated during 1000 client requests per minute did not exceed 3% and in fact was hard

to observe on monitored hosts. Additional memory usage of Mercury and Event Monitor was changing from 2 to 4 MB on each host.

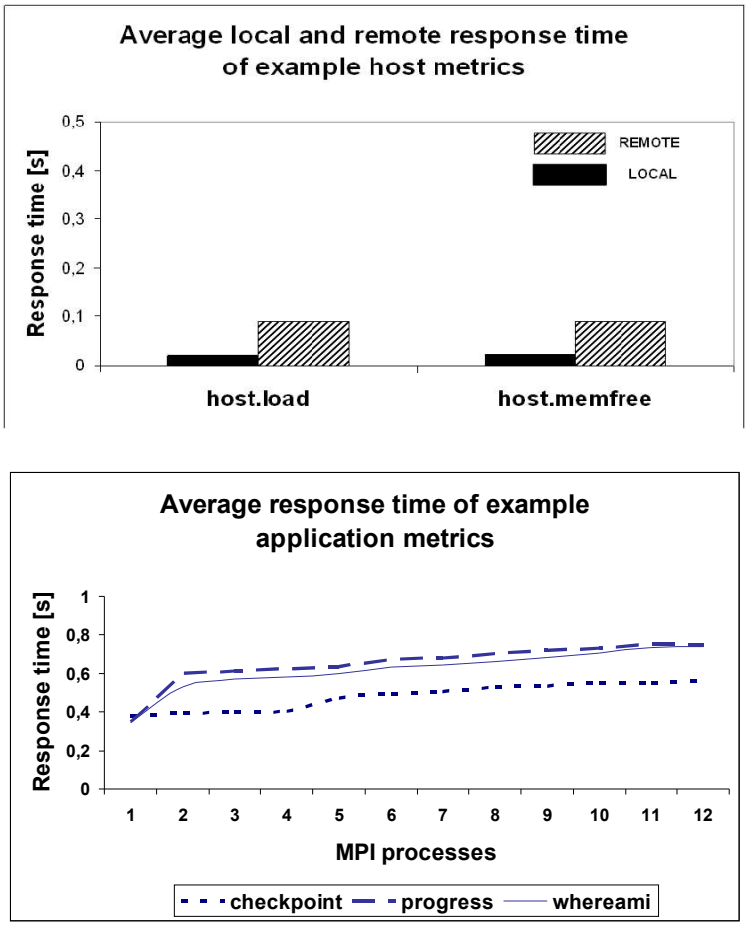


Fig 6. Response times of basic monitoring operations performed on Mercury and Event Monitor.

In our tests we have been constantly querying Mercury locally from many client tools and the average response time of all host metrics monitored on various hosts was stable and equaled approximately 18 ms. Remote response times as we expected were longer due to Internet network delays (70ms). The next figure shows us results of application oriented metrics which have been added in various testing MPI applications. The important outcome is that the response time (less than 1 second) did not increase significantly when more MPI processes were used, what is important especially to adopt monitoring capabilities for large scale experiments running on much bigger clusters.

All these performance tests have proved efficiency, scalability and low intrusiveness of both Mercury and Event Monitor and encouraged us for further research and development. Currently, as it was mentioned in section 5, Event Monitor works as an external application as far as Mercury's viewpoint is concerned but this does not restrict its functionality. However, in the future it may become more tightly integrated with the Mercury system (e.g. as a Mercury module) due to performance and maintenance reasons. To facilitate integration of Mercury and Event Monitor with external clients or grid middleware services, in particular GRMS, we have also developed the JEventmonitor-client package which provides a higher level interface as a simple wrapper

based on the low-level metric/control calls provided by Mercury. Additionally, to help application developers we have developed easy-to-use libraries which connect applications to Mercury and allow them to take advantage of mentioned monitoring capabilities.

Acknowledgment

Integration between GRMS and Mercury as well as performance and cost tests have been done in the scope of CoreGrid project at PSNC and SZTAKI labs. This project is founded by EU and aims at strengthening and advancing scientific and technological excellence in the area of Grid and Peer-to-Peer technologies.

References

1. http://www.gridlab.org
2. http://glite.web.cern.ch/glite/
3. http://www.globus.org
4. http:// www.gridlab.org/grms/
5. G, Gombás and Z. Balaton. "A Flexible Multi-level Grid Monitoring Architecture", In Proc. of 1st European Across Grids Conference, Santiago de Compostela, Spain, 2003. Volume 2970 of Lecture Notes in Computer Science, p. 214-221
6. K. Cooper et al., "New Grid Scheduling and Rescheduling Methods in the GrADS Project", In *Proceedings of Workshop for Next Generation Software* (held in conjunction with the IEEE International Parallel and Distributed Processing Symposium 2004), Santa Fe, New Mexico, April 2004
7. E. Huedo, R. Montero, and I. Llorente, "The Gri*d*Way Framework for Adaptive Scheduling and Execution on Grids", In *Proceedings of AGridM Workshop* (in conjunction with the 12th PACT Conference, New Orleans (USA)), Nova Science, October 2003.
8. S. Vadhiyar and J. Dongarra, "A Performance Oriented Migration Framework For The Grid", In *Proceedings of CCGrid*, *IEEE Computing Clusters and the Grid*, CCGrid 2003, Tokyo, Japan, May 12-15, 2003
9. "Improving Grid Level Throughput Using Job Migration and Rescheduling Techniques in GRMS. Scientific Programming", Krzysztof Kurowski, Bogdan Ludwiczak, Jarosław Nabrzyski, Ariel Oleksiak, Juliusz Pukacki, IOS Press. Amsterdam The Netherlands 12:4 (2004) 263-273
10. M. Gerndt et al., "Performance Tools for the Grid: State of the Art and Future", Research Report Series, Lehrstuhl fuer Rechnertechnik und Rechnerorganisation (LRR-TUM) Technische Universitaet Muenchen, Vol. 30, Shaker Verlag, ISBN 3-8322-2413-0, 2004
11. Serafeim Zanikolas and Rizos Sakellariou, "A Taxonomy of Grid Monitoring Systems", in Future Generation Computer Systems, volume 21, p.163-188, 2005, Elsevier, ISSN 0167-739X

Using High Level Petri-Nets for Describing and Analysing Hierarchical Grid Workflows

Martin Alt[1], Andreas Hoheisel[2], Hans-Werner Pohl[2], and Sergei Gorlatch[1]

[1] Westfälische Wilhelms-Universität Muenster, Germany
{mnalt|gorlatch}@uni-muenster.de
[2] Fraunhofer FIRST, Berlin, Germany
{andreas.hoheisel|hans.pohl}@first.fraunhofer.de

Abstract. In recent Grid computing environments, a common application programming model is to deploy often-used functionalities as remote services on high-performance Grid hosts, following the principles of a service-oriented Grid architecture. Complex applications are created by using several services and specifying a workflow between them. We discuss how the workflow of Grid applications can be described easily as a High-Level Petri Net (HLPN), in order to orchestrate and execute distributed applications on the Grid automatically.
Petri Nets provide an intuitive graphical workflow description, which is easier to use than script-based descriptions and more expressive than directed acyclic graphs (DAG). Furthermore, the Petri Net theory provides powerful analysis techniques that can be used to verify workflows for certain properties such as conflicts, deadlocks and liveness. In order to simplify the handling of complex and huge workflows, we introduce hierarchical Grid workflows, making use of the Petri Net refinement paradigm that allows to represent certain sub workflows by single graph elements.
We show how a complex application, the Barnes-Hut algorithm for N-Body simulation can be expressed as an hierarchical HLPN, using our platform-independent, XML-based Grid Workflow Description Language (GWorkflowDL). We discuss how the GWorkflowDL can be adapted to current Grid platforms, in particular to Java/RMI and the current WSRF standard.

1 Introduction

The service-oriented approach to Grid programming is to use remotely accessible services which are implemented on Grid hosts and provide commonly used functionality to applications running on clients. Popular examples of Grid middleware following the paradigm of the *Service-Oriented Architecture (SOA)* are the OGSI-compliant Globus Toolkit 3 [1] and the Web Services Resource Framework (WSRF) standard [2] with several implementations, such as Globus Toolkit 4 and WSRF.net.

Grid applications for service-based systems are usually composed of several services working together. An application developer has to decide which services offered by the Grid should be used in the application, and he has to specify the data and control flow between them. We will use the term *workflow* to refer to the automation of both – control and data flow.

In order to simplify Grid programming, it should be possible to describe an application workflow in a simple, intuitive way. Script-based workflow descriptions (e.g. GridAnt [3], BPEL4WS [4]) explicitly contain a set of specific workflow constructs, such as *sequence* or *while/do*, which are often hard to learn for unskilled users. Purely graph-based workflow descriptions have been proposed (e.g. for Symphony [5] or Condor's DAGman tool [6]) which are mostly based on Directed Acyclic Graphs (DAGs). Compared to script-based descriptions,

DAGs are easier to use and more intuitive: communications between different services are represented as arcs going from one service to another. However, DAGs offer only a limited expressiveness, so that it is often hard to describe complex workflows, e.g. loops cannot be expressed directly.

We propose a Grid Workflow Description Language (GWorkflowDL) [7] based on High-Level Petri Nets (HLPNs). HLPNs allow for nondeterministic and deterministic choice, simply by connecting several transitions to the same input place and annotating edges with conditions. Similarly, since DAGs only have a single node type, data flowing through the net cannot be modelled easily. In contrast, HLPNs make the state of the program execution explicit by tokens flowing through the net. The novelty of our approach is that we do not modify or extend the original HLPN model in order to describe services and control flow: we use the HLPN concept of edge expressions to assign a particular service to a transition, and we use conditions as the control flow mechanism. The resulting workflow description can be analysed for certain properties such as conflicts, deadlocks and liveness, using standard algorithms for HLPNs. We apply our HLPN-based language to a complex case study, the Barnes-Hut algorithm for N-body simulation.

The GWorkflowDL itself is platform-independent and provides platform-specific language extensions to adapt the generic workflow to a particular Grid platform. In this paper, we present two such extensions: for Java/RMI and for WSRF. The GWorkflowDL is intended to provide a common approach for the whole life-cycle of Grid applications, consisting of the workflow orchestration, scheduling, enactment, execution, and monitoring. The GWorkflowDL is currently the basis for the K-Wf Grid project [8], and the Java Grid programming system of the University of Muenster. The Fraunhofer Resource Grid [9] uses a similar approach, which is described in [10].

The structure of the paper is as follows: In the next section, we present the underlying Grid infrastructure and present our Grid workflow language. We discuss the workflow for the Barnes-Hut algorithm as a case-study in Section 3. In Section 4 we present the basic features of the Grid Workflow Description Language and the specific extensions for WSRF and Java/RMI platforms. We conclude our paper in the context of related work.

2 Using Petri Nets for Describing Workflows

Our graphical notation for Grid workflow is based on High-Level Petri Nets (Petri Nets with individual tokens), which allow to compute the value of output tokens of a transition based on the value of the input tokens. An introduction to the theoretical aspects of HLPNs can be found, e.g., in [11]. Van der Aalst and Kumar [12] give an overview of how to describe different workflow patterns using Petri Nets.

2.1 Workflow Elements

Petri Nets are directed graphs, with two distinct sets of nodes: *transitions* (represented by thick vertical lines or rectangles) and *places* (denoted by circles). Places and transitions are connected by directed edges. An edge from a place p to a transition t is called an *incoming edge* of t, and p is called *input place*. Outgoing edges and output places are defined accordingly. Each place can hold a number of individual *tokens* that represent data items flowing through the net. A transition is *enabled* if there is a token present at each of its input places. Enabled transitions can *fire*, consuming one token from each of the input places and putting a new token on each of the output places. The number and values of tokens each place holds is specified by the *marking* of the net. Consecutive markings are obtained by firing transitions.

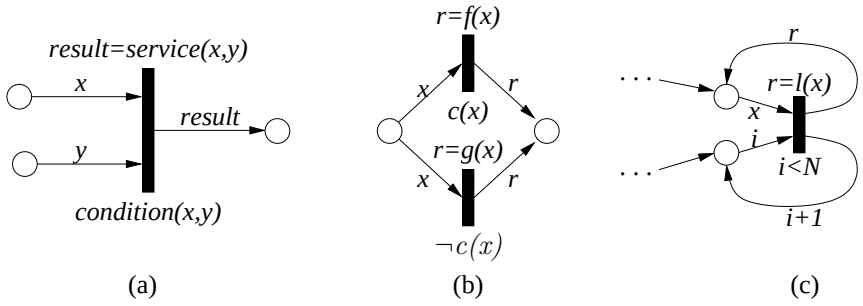


Fig. 1. HLPNs for single services, branches and loops.

Each edge can be assigned an *edge expression*. For incoming edges, variable names are used, assigning token values obtained through this edge to a specific variable. Additionally, each transition can have a set of boolean *condition* functions. A transition can only fire if all of its conditions evaluate to true for the input tokens. As an example, Fig. 1 (a), shows a transition representing a *service* which receives input parameters x and y and produces a result value.

The service name is written above the transition, variables for the formal parameters and results are represented as places, with parameter names (e.g. x and y in the Figure) shown as edge expressions on incoming edges. The edge expressions for outgoing edges specify which value should be placed on the corresponding output place when the transition fires ($result$ in the Figure). Usually, this is the result of the invoked service, but it can also be an error value.

In addition to the service itself, a set of *conditions* may be associated with a transition (shown beneath the transition). Conditions can be used to check whether the input data of the service meets certain requirements. Additionally, the application programmer can use conditions to realise standard control flow structures, such as conditions and loops. For example, consider the net shown in Fig. 1 (b): an input place is connected to two concurring transitions f and g. If a condition c is true, then f is executed, else g. Similarily, loops can be realised as shown in Fig. 1 (c) where the loop body l is executed as long as condition variable i is less than N. Note that edge expressions do not necessarily have to be related to the service's input and output variables: they can also be used to perform simple computations, as shown in the loop example in Fig. 1 (c) where i is only used as a counter and not passed to the service l itself.

Besides places and edges for input and output data of transitions, the application developer has the possibility to introduce additional *control places* to the graph. A control place holds simple tokens which do not carry any value. Accordingly, input edges connecting a control place to a transition (*control edges*) have no associated variables, and the token values are not used as parameters for the associated service. Control edges just synchronise the firing of a transition with the corresponding control place.

2.2 Hierarchical workflows

In order to simplify the design of complex workflows, our workflow description allows to use *composite transitions*, which can be used as normal transitions in a workflow but represent subworkflows themselves. For example, transition t in Fig. 2 left is a composite transition, representing a workflow consisting of transitions t_1, t_2 and t_3. The subworkflow is connected to the outside workflow by a set of incoming and outgoing edges and places (x_1, x_2, r_1, and r_2 in the Figure). When a composite transition is chosen to fire during workflow execution, the corresponding subnet is executed.

When replacing a single transition by a whole subworkflow, we make use of the Petri Net refinement theory, which defines some constraints in order to assure that the embedded

Fig. 2. Hierarchical workflows: a composite transition representing a subworkflow (left) and a semantically equivalent HLPN without composite transitions (right).

sub Petri Net has no causal influence to the top level net, and that the refined net remains semantically equivalent. Essentially the refinement has to meet the following requirements:

- The number and direction of incoming and outgoing edges – which connect the composite transition and the embedded subworkflow with the top level Petri Net – have to remain the same before and after the refinement.
- The subworkflow must consume and produce the same amount of tokens as the original transition.
- The subworkflow must be free of contact, i.e., the capacity of the places does not prevent a transition from firing.

Following these requirements makes it feasable to compose hierarchical workflows by means of Petri Net refinement, however, one have to be aware of two basic differences between traditional 'flat' Petri nets and hierarchical Petri nets as introduced in this paper:

1. Normally, transitions fire instantaneously, i.e., the tokens on the input places are consumed and new tokens are put on the output places within a single step. In general, this is not the case if a transition represents a sub Petri net, which processes the tokens during several steps. In our approach, input tokens as well as output tokens may not be consumed and produced within the same step.
2. It is not trivial to introduce new edges or conditions to a transition that represents a sub Petri Net. In order to simplify this process one may introduce single entry and exit transitions, such as shown in Fig. 2 (right).

For example, consider Fig. 2 left, where transition t_1 can fire independently of transitions t_2 and t_3. If they are grouped together in a composite transition t then x_1 and x_2 are incoming edges of a single transition, which can only fire if there is data available on both input places. Thus, the semantics is the same as shown in Fig. 2 right: in addition to the transitions t_1, t_2, and t_3, there are two more transitions I and O which synchronise the input and output of the subnet.

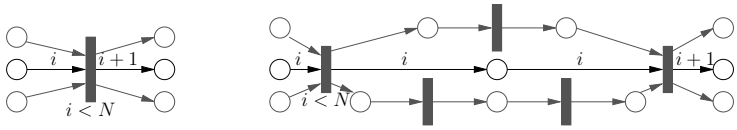


Fig. 3. User-defined edges and conditions on composite transitions.

Using composite transitions, it is possible to assign conditions to subworkflows. As an example, a loop with a composite transition as a body is shown in Fig. 3 left: an input edge

i for counting and a condition $i < n$, to check the number of iterations are added to the composite transition. This is equivalent to the net shown in Fig. 3 right: the conditions on the input edges are checked at transition I and tokens are then passed to output transition O, to which the outgoing edge is connected.

3 Case study: The Barnes-Hut Algorithm

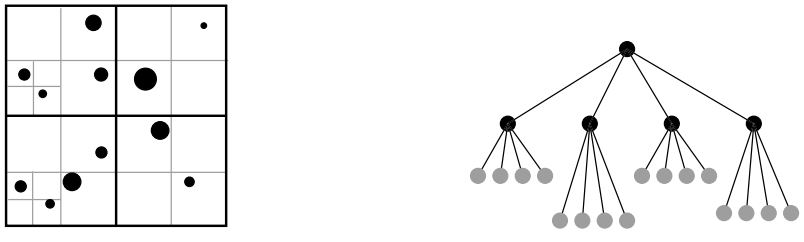
We will now discuss, how a comparatively complex application can be expressed as a HLPN. The *Barnes-Hut (BH)* algorithm [13] is a widely used approach to computing force interactions of bodies (particles) based on their mass and position in space, e.g. in astrophysical simulations. At each timestep, the pairwise interactions of all bodies have to be calculated, which implies a computational complexity of $O(n^2)$ for n bodies. The BH algorithm reduces the complexity to $O(n \cdot logn)$, by grouping distant particles: for a single particle in the BH algorithm, distant groups of particles are considered as a single object if the ratio between the spatial extent of the particle group and the distance to the group is smaller than a simulation-specific coefficient θ (chosen by the user).

For an efficient access to the huge amount of possible groups in a simulation space with a large number of objects, the BH algorithm subdivides the 3D simulation space using a hierarchical *octree* with eight child cubes for each node (or *quadtree* for the 2D case). The tree's leaves contain single particles, parental nodes represent the particle group of all child nodes and contain the group's centre and aggregated mass. The force calculation of a single particle then is performed by a depth-first traversal of the tree. Fig. 4(a) and 4(b) depict an example partition and the resulting quadtree for the 2D case (see [13] for further details and complexity considerations).

We have implemented the BH algorithm for the Grid using the Java-based Grid system developed at the University of Muenster [14], using a set of generic, high-level services. Details about the implementation can be found in [15]. We will now show, how the workflow of this complex Grid application can be expressed easily as a HLPN.

3.1 Barnes-Hut: Workflow for a single Timestep

The computations for one timestep of the algorihm are decomposed into a workflow containing six services as shown in Fig. 5, which correspond to the following steps of the algorithm:
3. Calculation of the spatial boundary of the simulation space: In order to build the tree, it is necessary to know the boundaries of the universe to be simulated. This is done using service *compBB* which produces a bounding box *bb* as output. Note that this bounding box is copied to two output places for use by two other services. Also, the array of particles *part* received as input is copied to a third output place, as it is also used by the next service.



(a) Space partitioning (b) Quadtree

Fig. 4. Barnes-Hut octree partition of the simulation space

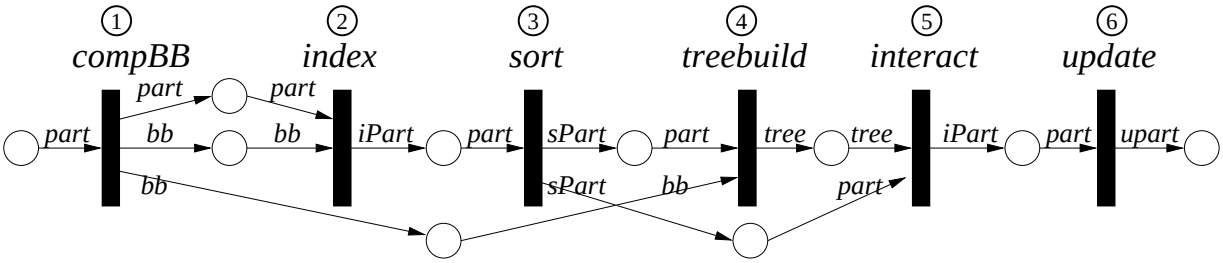


Fig. 5. HLPN for one iteration of the BH algorithm.

4. Indexing: In order to group particles which are nearby in space, the particle array must be sorted so that nearby particles are also at nearby positions in the particle array. As a first step for sorting, an index is computed for each particle, based on its spatial location (see [15] for details), using service *index*. The result *iPart* is a particle array, where each particle has an index associated with it.

5. Sorting: The particles are then sorted in ascending order of the index computed in the previous step using service *sort*. The resulting sorted particle array *sPart* is used as input for two other services and thus copied to two different output places.

6. Building the octrees: This step builds the octree representation of the universe using service *treebuild*. The resulting tree is used to group particles for efficient access.

7. Force computation: In this step, the interactions of each particle with all others is computed by service *interact*. For each particle in *sPart*, the octree *tree* is traversed and the force effects of the current node is added to the velocity vector of the particle if the node represents a group that is small enough or far enough away. If this criterion is not yet met, then the eight child nodes are processed recursively.

8. Particle update: Finally, in the *update* service, for each particle, the current particle's position is updated according to the forces computed in the previous step.

Each of the services can be executed remotely on parallel high-performance Grid servers, using a Java-based programming system for the Grid, as described in [14].

3.2 Barnes-Hut: Workflow for Loop

The workflow for a single timestep described above is executed iteratively to evolve the simulated universe for a user-defined amount of time. The corresponding workflow is shown in Fig. 6.

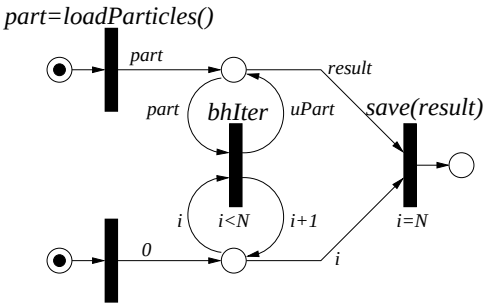


Fig. 6. HLPN for BH loop

The single-iteration workflow is encapsulated in a composite transition *bhIter*, which is executed in a bounded loop. Before the algorithm starts, initial particle positions and velocities are loaded using a service *loadParticles*. Also, the iteration counter is initialised with 0, using an empty transition, which is not associated with any service, but places a 0 on its output place when executed. Finally, transition *save* is used to save the result after N timesteps.

4 Grid Workflow Description Language (GWorkflowDL)

The language GWorkflowDL is being developed as an XML-based language for Grid workflows, based on HLPNs as described in the previous section. It consists of a generic part, used to define the structure of the workflow, and a platform-specific part (*extension*) defining how to execute the workflow in the context of specific Grid computing platforms.

4.1 GWorkflowDL – XML Schema

Figure 7 graphically represents the XML Schema of GWorkflowDL. The root element is called `<workflow>`: it contains the optional element `<description>` with a human-readable description of the workflow, and several occurrences of the elements `<transition>` and `<place>` that define the Petri Net of the workflow.

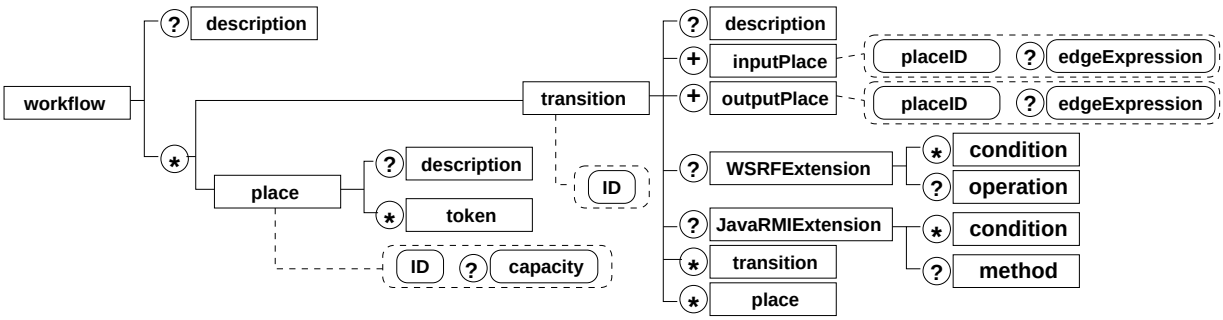


Fig. 7. Graphical representation of the XML schema for GWorkflowDL. Boxes denote elements, rounded boxes represent attributes. Legend: $? = 0, 1; * = 0, 1, 2, \ldots; + = 1, 2, 3, \ldots$

The element `<transition>` may be extended by platform-specific child elements, such as `<WSRFExtension>` and `<JavaRMIExtension>`, which represent special mappings of transitions onto particular Grid platforms, or it may contain a set of `<transition>` and `<place>` elements if the transition is composite. Elements `<inputPlace>` and `<outputPlace>` define the edges of the net. Edge expressions are represented as attribute `edgeExpression` of `InputPlace` and `OutputPlace` tags.

4.2 GWorkflowDL – Platform Extensions

To adapt a generic workflow description to a particular Grid computing platform, we use *extensions*, which describe the meaning of a generic net in the context of a particular platform. We will now present two example extensions, for WSRF and Java/RMI. Platform extensions define: (1) the platform-specific service to be invoked; (2) how conditions are evaluated; (3) how edge expressions are evaluated. The GWorkflowDL document for the a sort service that sorts integers in ascending order is as follows:

```
<workflow>
 <place ID="P1"/> <place ID="P2"/> <place ID="R"/>
 <transition ID="ZIP">
  <inputPlace placeID="P1" edgeExpression="x"/>
  <outputPlace placeID="R" edgeExpression="result"/>
  <JavaRMIExtension>
   <method name="result = Sort.execute(x)"/>
   <condition name="x != null"/>
  </JavaRMIExtension>
  <WSRFExtension>
   <operation name="sort" owl="gom.kwfgrid.net/sort.xml">
    <WSClassOperation name="sortI" owl="kwfgrid.net/sortI.xml">
     <WSOperation name="sortI@first" owl="first.fhg.de/sort.xml"/>
     <WSOperation name="sortI@iisas" owl="savba.sk/sort.xml"
                  selected="true"/>
     <WSOperation name="sortI@cyfro" owl="agh.edu.pl/sortI.xml"/>
   </WSClassOperation> </operation>
  </WSRFExtension>
 </transition>
</workflow>
```

Note that this code is not intended to be written by the programmer, but rather can be generated automatically from Java or WSDL interfaces, or by workflow orchestration tools for combining several services. For the Java extension, *edge expressions* assign variable names, and the *method* element captures services and describes how the methods and conditions should be called. The *condition* elements provide conditions which contain a Java expression that depends on the input variables and yields a boolean value. The code also shows the abbreviated GWorkflowDL representation of the sort service using the `WSRFExtension` as used in K-Wf Grid [8]. The *edge expressions* and *conditions* may be specified as XPath queries. The *operation* element captures several levels of abstraction of web service operations: *operation* describes a very abtract operation without any details, *WSClassOperation* specifies a operation on specific class of Web Services described by their interfaces and functionality, and *WSOperation* are the concrete instances of Web Service operations that match the class. The *owl* attribute links to external semantic descriptions.

4.3 Workflow Orchestration and Execution

We assume a Grid system architecture as shown in Fig. 8, where application programs are constructed using a set of services which are implemented on remote high-performance hosts. Services are invoked from a client on remote Grid hosts using a remote method invocation mechanism such as Java/RMI or SOAP.

To design a Grid program, the application developer first selects the services required for the application and creates an abstract workflow, such as the net shown in Fig. 5. The resulting abstract workflow description can already be analysed for certain properties such as deadlocks and liveness, using standard algorithms for HLPNs (see e.g. [16]).

After selecting an appropriate service-based Grid computing platform, the application developer has to adapt the abstract Petri Net to a particular platform by assigning particular services and platform-specific edge expressions to transitions. E.g., for the Java platform, a Java method of a remote interface is assigned to each transition, and variable names are assigned to the input and output edges. The resulting specific HLPN for the desired workflow can then be executed on the Grid by assigning an executing host to each service, either manually (to

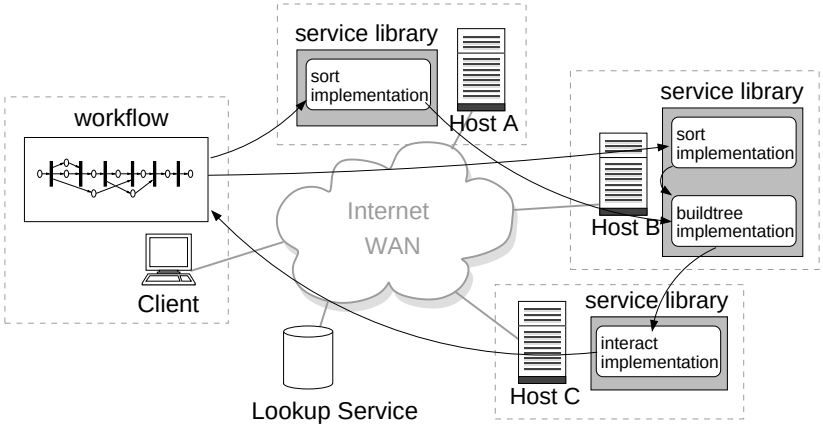


Fig. 8. Prototype Grid architecture.

execute the application on a user-selected set of hosts) or automatically, using a scheduling strategy to select hosts.

Execution then starts by selecting an enabled transition. The tokens on the input places are collected and the transition's conditions are evaluated. If all conditions yield true, the corresponding service is invoked, with the data related to the tokens on input places as input parameters. The result is then placed as token on the output places. If at least one condition evaluates to false, the input tokens are returned to their respective input places. Then the next enabled transition is selected. This process continues until each terminal place holds at least one token, or no enabled transitions exist.

5 Conclusions

We have presented our approach for expressing Grid application workflows as High-Level Petri Nets and described GWorkflowDL, an XML-based language for specifying Grid workflows. Petri nets are widely used for modelling and analysing business workflows in workflow management systems (e.g. [17]). The use of Petri Nets for Grid workflow has first been proposed as Grid Job Definition Language (GJobDL) [10] for job-based Grid systems, where a Grid application is composed of several *atomic Grid jobs* which are sent to the hosts for execution. The GJobDL language is similar to the GWorkflowDL, but it uses a modified HLPN where transitions contain input and output *ports*, representing parameters and results, and edges connect places to ports instead of transitions.

In contrast, our GWorkflowDL specifies Grid workflow as a standard HLPN (as defined e.g. in [11]), using conditions for control flow and edge expressions to assign parameters and results. Adhering more strictly to the standard model of HLPNs allows us to make use of standard algorithms for analysing Nets, e.g. for deadlocks.

The HLPN representation of a workflow serves four main purposes: (1) It is an intuitive graphical description of the program, making communication between services explicit and allowing users to develop programs graphically without having to learn a specific workflow language. (2) Because applications are developed as unmodified HLPNs, the application's HLPN can be used for analysis and formal reasoning based on the results of previous research in High-Level Petri Nets. (3) The same GWorkflowDL description can be used to monitor and inspect running and finished workflows. (4) Because the GWorkflowDL is divided into an abstract and a platform-specific part, it can be used with different service implementations and Grid platforms.

As future work, we plan to implement a set of tools for workflow orchestration, execution, monitoring, and analysis, based on the GWorkflowDL. In particular, we intend to implement performance prediction of Grid applications by using time values as tokens and service functions that assign the expected performance of particular services (which can be obtained using an approach discussed in [18]) to the input tokens.

Acknowledgements

Our work is supported in part by the European Union through the IST-2002-004265 Network of Excellence CoreGRID and the IST-2002-511385 project K-WfGrid.

References

1. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The physiology of the grid: An open grid services architecture for distributed systems integration. In: Open Grid Service Infrastructure WG, Global Grid Forum. (2002)
2. Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W.: The WS-Resource Framework (2004) http://www.globus.org/wsrf/.
3. von Laszewski, G., Alunkal, B., Amin, K., Hampton, S., Nijsure, S.: GridAnt – client-side workflow management with Ant. http://www-unix.globus.org/cog/projects/gridant/ (2002)
4. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., Weerawarana, S.: Business process execution language for web services version 1.1. Technical report, BEA Systems, IBM, Microsoft, SAP AG and Siebel Systems (2003)
5. Lorch, M.: Symphony – A Java-based Composition and Manipulation Framework for Computational Grids. PhD thesis, University of Applied Sciences in Albstadt-Sigmaringen, Germany (2002)
6. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: The Condor experience. Concurrency and Computation: Practice and Experience (2004)
7. Hoheisel, A., H.-W.Pohl: Documentation of the Grid Workflow Description Language toolbox. http://fhrg.first.fraunhofer.de/kwfgrid/gworkflowdl/docs/ (2005)
8. K-Wf Grid consortium: K-Wf Grid homepage. http://www.kwfgrid.net/ (2005)
9. Fraunhofer Gesellschaft: Fraunhofer Resource Grid homepage. http://www.fhrg.fraunhofer.de/ (2005)
10. Hoheisel, A., Der, U.: An XML-based framework for loosely coupled applications on grid environments. In Sloot, P., ed.: ICCS 2003. Number 2657 in Lecture Notes in Computer Science, Springer-Verlag (2003) 245–254
11. Jensen, K.: An introduction to the theoretical aspects of Coloured Petri Nets. In de Bakker, J., de Roever, W.P., Rozenberg, G., eds.: A Decade of Concurrency. Volume 803 of Lecture Notes in Computer Science., Springer-Verlag (1994) 230–272
12. van der Aalst, W.M.P., Kumar, A.: Xml based schema definition for support of inter-organizational workflow. University of Colorado and University of Eindhoven report (2000)
13. Barnes, J.E., Hut, P.: A hierarchical $O(N \log N)$ force-calculation algorithm. Nature **324** (1986) 446–449
14. Alt, M., Bischof, H., Gorlatch, S.: Algorithm design and performance prediction in a Java-based Grid system with skeletons. In Monien, B., Feldmann, R., eds.: Euro-Par 2002. Volume 2400 of Lecture Notes in Computer Science., Springer-Verlag (2002) 899–906
15. Alt, M., Müller, J., Gorlatch, S.: Towards high-level grid programming and load-balancing: A Barnes-Hut case study. In Cunha, J.C., Medeiros, P.D., eds.: Euro-Par 2005 Parallel Processing. Volume 3648 of Lecture Notes in Computer Science., Springer-Verlag (2005) 391–400
16. Girault, C., Valk, R., eds.: Petri Nets for Systems Engineering. Springer-Verlag (2003)
17. van der Aalst, W.: The application of Petri Nets to workflow management. The Journal of Circuits, Systems and Computers **8** (1998) 21–66
18. Alt, M., Bischof, H., Gorlatch, S.: Program development for computational Grids using skeletons and performance prediction. Parallel Processing Letters **12** (2002) 157–174

Issues about the Integration of Passive and Active Monitoring for Grid Networks

S. Andreozzi[2], D. Antoniades[1], A. Ciuffoletti[2], A. Ghiselli[2], E.P. Markatos[1], M. Polychronakis[1], P. Trimintzios[1]

[1] FORTH-ICS, P.O. Box 1385 – 71110, Heraklion, GREECE,
{ptrim,mikepo,danton,markatos}@ics.forth.gr
[2] CNAF-INFN, Via Berti Pichat 6/2 – 40126, Bologna, ITALY
augusto@di.unipi.it,{sergio.andreozzi,antonia.ghiselli}@cnaf.infn.it

Abstract. We discuss the integration of passive and active techniques in a Grid monitoring system. We show the advantages obtained by using the same domain-oriented overlay network to organize both kinds of monitoring.

1 Introduction

Grid applications require Storage, Computing, and Communication resources, and need to know the characteristics of such resources in order to setup an optimal execution environment. At present, Storage and Computing resources monitoring is sufficiently precise, and is translated into database schemas that are used for early experiments in system resources optimization. In contrast, monitoring of Communication resources is at an early stage, due to the the complexity of the infrastructure to monitor and of the monitoring activity.

According to the Global Grid Forum (GGF) schema [3], the management of network measurements (which we call *observations*) is divided into three distinct activities: their *production*, their *publication*, and their *utilization*. Here, we focus on the infrastructure related to *production* and *publication*.

Our primary concern is scalability when producers are increasing in number and monitoring data output: in order to limit the quantity of observations that need to be published, we use a *domain-oriented* overlay network. Under this light, in Section 2 we describe alternative techniques for network monitoring, and we devise an hybrid network monitoring architecture. Section 3 addresses a number of security and privacy issues related to such architecture.

2 Classification of Monitoring Approaches and Techniques

In this section we classify monitoring approaches according with two criteria: the first criterion distinguishes *path* and *link* granularity for network monitoring, while the second classification divides monitoring tools into *active* and *passive* ones.

2.1 Finding a Compromise Between Link and Path Monitoring

One issue that emerges when considering network monitoring is related to its granularity. We envision two main alternatives:

single link - it gives the view from a single observation point. It is good for maintainers, which need a fine grained view of the network in order to localize a problem, but inappropriate for Grid-aware applications, that may need end-to-end observations. Note that correlation of the information from multiple single links may provide monitoring metrics appropriate for some Grid applications.

end-to-end path - it gives a view of the system that is filtered through routing: this may be sometimes confusing for maintainers, but is appropriate for Grid aware applications.

However, the scalability of the two approaches is dramatically different: let N be the number of resources in the system. A link oriented monitoring system grows with $O(N)$, since the Grid can be assimilated to a bounded degree graph. In a path-oriented approach, the address space is $O(N^2)$, since, as a general rule, each resource has a distinct path to any other resource.

This consideration seems to exclude the adoption of a end-to-end path approach, but there are other problems with the single-link approach:

- edges of a link are often black boxes that contain proprietary software: there may be no way to modify or add code for monitoring purposes, or even to simply access the stored data;
- deriving an end-to-end path performance metric from single-link observations requires two critical steps: to reconstruct the link sequence, and, even more problematic, to obtain time correlated path performance compositions from single-link observations;

We conclude that each approach exhibits severe drawbacks, and we propose a compromise: we introduce an overlay network that cluters network services into *domains*, and restricts monitoring to inter-domain paths. Such a strategy, which resembles the BGP/OSPF dichotomy in the Internet, finds a compromise between the two extreme design strategies outlined above:

- like an *end-to-end path strategy*, it offers Grid oriented applications a valuable insight of the path connecting two resources. However, such insight does not include the performance of the local network (which usually outperforms inter-domain paths), and the address space is still $O(N^2)$, but now N stands for the number of domains, which should be significantly smaller than the number of resources;
- like a *single link strategy*, it provides the maintainers with a reasonable localization of a problem. As for accounting, as long as domains are mapped to administrative entities, it gives sufficient information to account resource utilization.

In essence, a *domain-oriented* approach limits the complexity of the address space into a range that is already managed by routing algorithms, avoids path reconstruction, and has a granularity that is compatible with relevant tasks. The overlay view it introduces cannot be derived from a pre-existent structure: the Domain Name System (DNS) structure is not adequate to map monitoring domains, since the same DNS subnetwork may in principle contain several monitoring domains, and a domain may overlap several DNS subnetworks. The overlay network (or *domain partition*) must be separately designed, maintained, and made available to users, as explained in section 2.5.

2.2 Passive and Active Monitoring Techniques

Another classification scheme distinguishes between active and passive monitoring. The definition itself is slippery, and often a matter of discussion. For our purpose, we adopt the following classification criterion:

> *a monitoring tool is classified as* active *if its measurements are based on traffic it induces into the network, otherwise it is* passive.

Passive monitoring tools can give an extremely detailed view of the performance of the network, while active tools return a response that combines several performance figures.

As a general rule, effective network monitoring should exploit both kinds of tools:

- an active approach is more effective to monitor network sanity;
- an active approach is suitable for application oriented observations (like jitter, when related to multimedia applications);
- a passive approach is appropriate to monitor gross connectivity metrics, like throughput;
- a passive approach is needed for accounting purposes.

In the following, we discuss both passive and active monitoring in the context of monitoring data *production* for Grid infrastructures.

2.3 Passive Network Monitoring for Grid Infrastructures

Passive network monitoring techniques analyze network traffic by capturing and examining individual packets passing through the monitored link, allowing for fine-grained operations, such as deep packet inspection [1].

Figure 1 illustrates a high-level view of a distributed passive network monitoring infrastructure. Monitoring sensors are distributed across several domains, here considered for simplicity as Internet Autonomous Systems (AS). Each sensor may monitor the link between the domain and the Internet (as in AS 1 and 3), or an internal link of a local sub-network (as in AS 2). An authorized user, who may not be located in any of the participating Autonomous Systems, can

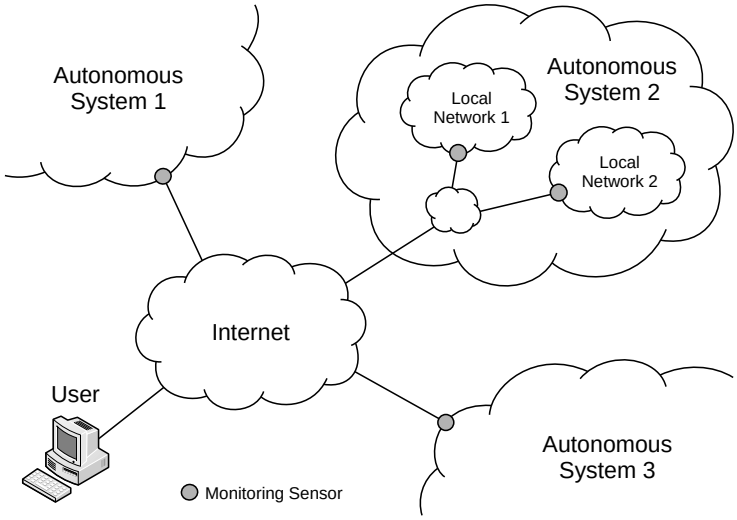


Fig. 1. A high-level view of a distributed passive network monitoring infrastructure.

run monitoring applications that require the involvement of an arbitrary number of the available monitoring sensors.

A passive monitoring infrastructure, either local or distributed, can be used to derive several connectivity performance metrics: we enlist some of these metrics, classifying them based on the number of passive monitoring observation points required to derive them.

Metrics Using a Single Observation Point

- *Network-level Round-Trip Time* (RTT) is one of the simplest network connectivity metrics, and can be easily measured using active monitoring tools like for example `ping`. However, it is also possible to measure RTT using solely passive monitoring techniques, based on the time difference between the `SYN` and `ACK` packets exchanged during the three-way handshake of a TCP connection.
- *Application-level Round-Trip Time* is measured, for instance, as the lapse between the observation of a request and of the relevant reply (see also EtE [6]).
- *Throughput*: passive monitoring can provide traffic throughput metrics at varying levels of granularity: the aggregate throughput provides an indication for the current utilization of the monitored link, while fine-grained per-flow measurements can be used to observe the throughput achieved by specific applications (see also [8]).
- *Retransmitted Packets*: the amount of retransmitted packets provides a good indication of the quality of a path.
- *Packet Reordering*: such events, as reported in [7], degrade application throughput. The percentage of reordered packets is obtained observing the sequence field in the header of incoming TCP packets.

Metrics Using Multiple Observation Points

- *One-Way Delay and Jitter*: OWD can be measured using two passive monitors with synchronized clocks located at the source and the destination. One way delay variation (or *jitter*) can also be computed.
- *Packet Loss Ratio*: this metric can be measured using two cooperating monitors at the source and the destination, keeping track of the packets sent but not received by the destination after a timeout period.
- *Service Availability*: a `SYN` packet without a `SYN-ACK` response indicates a refused connection, which gives an indication of the availability of a particular domain/service.

2.4 Active Monitoring for Grid Infrastructures

Active tools induce a test traffic benchmark into the Grid connectivity infrastructure, and observe the behavior of the network. As a general rule, one end (the *probe*) generates a specific traffic pattern, while the other (the *target*) cooperates by returning some sort of feedback: the `ping` tool is a well known representative of this category.

Disregarding the characteristics of the benchmark, an active monitoring tool reports a view of the network that is near to the needs of the application: for instance, a `ping` message that uses the Internet Control Message Protocol (ICMP) gives an indication of raw transmission times, useful for applications such as multimedia streaming. A `ping` that uses `UDP` packets or a short `ftp` session may be used to gather the necessary information for optimal file transfers. Since active tools report the same network performance that the application will observe, their results are readily usable by Grid-aware applications that want to optimize their performance.

The coordination activity associated to active monitoring is minimal: this is relevant for a dynamic entity, such as a Grid, where join and leave events are frequent. A new resource that joins the Grid enters the monitoring activity simply by starting its *probe* and *target* related activities. However, join and leave activities introduce security problems, which are further addressed in Section 3.

Most of the statistics collected by active tools have a local relevance, and need not be transmitted elsewhere: as a general rule, they are used by applications that run in the domain where the probe resides. A distributed *publication* engine may take advantage of that, exporting to the global view only those observations that are requested by remote *consumers*.

Network performance statistics that can be observed using active monitoring techniques can be divided into two categories:

packet oriented: related to the behavior induced by single packet transmissions between the measurement points. Besides RTT, appropriate probes allow for the observation of `TCP` connection setup characteristics and one-way figures of packet delay and packet delay variation;

stream oriented: related to the behavior induced by a sequence of packets with given characteristics. Such characteristics may include the specification of the timing and the lenght of the packet stream, as well as the content of individual packets. Examples of such streams are an `ftp` transfer of a randomly generated file of given length, or a back-to-back sequence of `UDP` packets.

A relevant feature shared by active monitoring tools is the ability to detect the presence of a resource, disregarding if it is used or not, since they require an active participation of all actors (probe, target and network). This not only helps fault tolerance, but may also simplify the maintenance of the Grid layout, which is needed by Grid-aware applications.

Since active monitoring consumes some resources, security rules should limit the impact of malicious uses of such tools: this issue is also covered in Section 3.

2.5 The Domain Overlay Database

The domain overlay database is a cornerstone of our monitoring system: the content of such a database reflects the *domain-oriented* view of the Grid.

The GlueDomains [5],[4] prototype serves as a starting point for our study. GlueDomains supports the network monitoring activity of the prototype Grid infrastructure of INFN, the Italian Institute for Nuclear Physics. GlueDomains follows a *domain-oriented* approach, as defined above. Monitoring activity results are published using the Globus Monitoring and Discovery System (MDS) [9]. MDS is the information services component of the Globus Toolkit that provides information about the available resources on the Grid and their status, and is rendered through the GridICE [2] toolset.

The domain overlay maps Grid resources into domains, and introduces further concepts that are specific to the task of representing the monitoring activity. In order to represent such an overlay view, we use the Unified Model Language (UML) graph outlined in Figure 2. The classes that represent Grid resources are the following:

Edge Service: it is a superclass that represents a resource that does not consist of connectivity, but is reached through connectivity.

Network Service: represents the interconnection between two Domains. Its attributes include a class, corresponding to the offered service class, and a statement of expected connectivity.

Theodolite Service: a Theodolite Service monitors a number of Network Elements. In GlueDomains, theodolites perform *active* network monitoring.

The following classes represent aggregations of services:

Domain: represents the partitions that compose the Grid. Its attributes include the service class offered by its fabric.

Multihome: represents an aggregation of Edge Services that share the same hardware support, but are accessible through distinct interfaces.

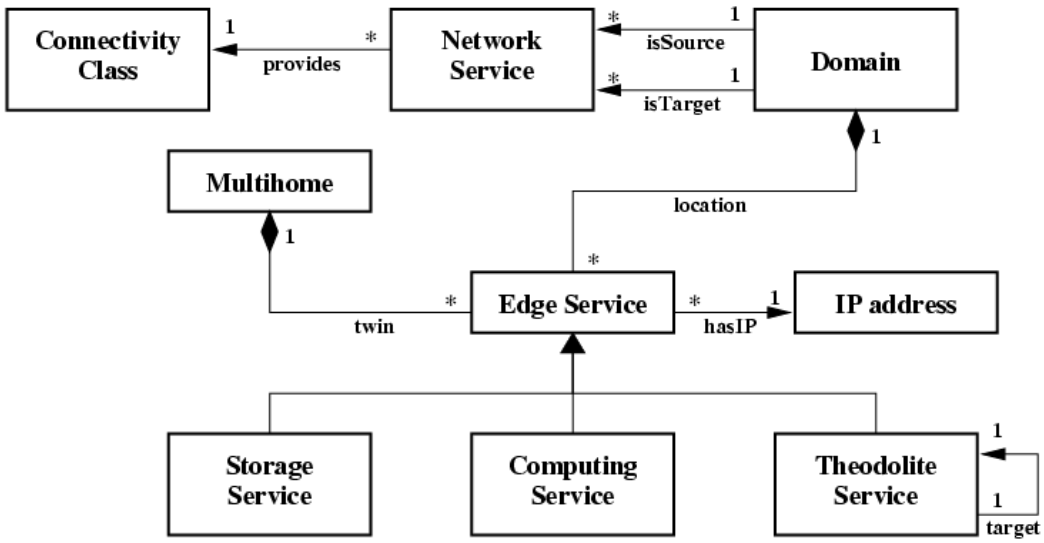


Fig. 2. The UML diagram of the topology database with domain partitioning

The description of the overlay network using the above classes is made available through a *topology database*, which is used by the *publication* engine in order to associate observations to network services.

Observations collected by *active* monitoring tools are associated to a network service based on the location of the theodolites. Observations collected by *passive* traffic observers are associated to a specific network service using basic attributes (like source and destination IP address, service class, etc.) of the packets captured by such devices. The knowledge of theodolites as hosts *relevant* from the point of view of network monitoring may indicate which packets are more significant, thus opening the way to the cooperation between theodolites and passive traffic observers.

2.6 Description of Monitoring Activities

Also relevant to the management of the monitoring activity is its description. In order to limit human intervention to the design and deployment of the network monitoring infrastructure, the description of the monitoring activity should be available to devices that contribute to this task, also considering the possibility of *self-organization* of such activity.

In the case of GlueDomains, theodolite services are the agents of monitoring configuration. The UML model shown in Figure 3 is centered around such entity, and describes the structure of the *monitoring database*.

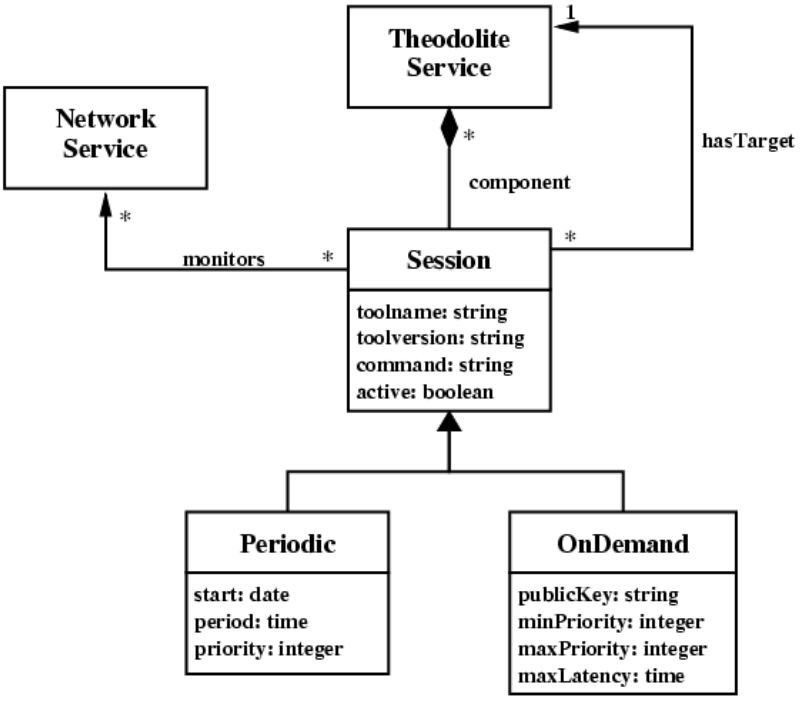


Fig. 3. The UML diagram of the monitoring database

Active monitoring is organized into *sessions*, each associated to a theodolite and to a monitored network service. The description of the monitoring session indicates a monitoring tool and its configuration. Passive monitoring is represented by specific session classes, and the theodolite will instruct remote passive monitoring devices about the required activity. An authentication mechanism avoids unauthorized use of passive monitoring devices.

3 Security and Privacy

A large-scale network monitoring infrastructure is exposed to several threats: each component should be able to ensure an appropriate degree of security, depending on the role it plays.

Monitoring sensors hosting passive or active tools may become targets of coordinated Denial of Service (DoS) attacks, aiming to prevent legitimate users from receiving a service with acceptable performance, or sophisticated intrusion attempts, aiming to compromise the monitoring hosts. Being exposed to the public Internet, monitoring sensors should have a rigorous security configuration in order to preserve the confidentiality of the monitored network, and resist to attacks that aim to compromise it.

The security enforcement strategy is slightly different for active and passive monitoring tools. In the case of passive monitoring tools, the monitoring host

should ensure the identity and the capabilities associated with a host submitting a request. Such a request may result to the activation of a given packet filter, or to the retrieval of the results of the monitoring activity. Each passive sensor should be equipped with a firewall, configured using a conservative policy that selectively allows inbound traffic according with accepted requests, and dropping inbound traffic from any other source. One option is to consider that only theodolite services, whose credentials (e.g., their public keys) are recorded in the monitoring database, are able to access passive sensor configuration, and therefore dynamically configure its firewall. Theodolite capabilities may vary according to a specific monitoring strategy.

In the case of active monitoring tools, the target is exposed to DoS attacks, consisting in submitting benchmark traffic from unauthorized, and possibly malicious, sources. One should distinguish between tools that are mainly used for discovery, and those that are used for monitoring purposes. The former should be designed as lightweight as possible, for instance consisting of a predetermined ping pattern: firewall on probe side shouldn't mask such packets, unless their source is reliably detected as threatening. The latter might result to rather resource consuming patterns, and the probe should filter packets according to an IP based strategy: such a configuration would be based on the content of the monitoring database.

Both passive and active monitoring tools have in common the need of ensuring an adequate degree of *confidentiality*. In fact, data transfers through TCP are unprotected against eavesdropping from third-parties that have access to the transmitted packets, since they can reconstruct the TCP stream and recover the transferred data. This would allow an adversary to record control messages, forge them, and replay them in order to access a monitoring sensor and impersonate a legitimate user. For protection against such threats, communication between the monitoring applications and a remote sensors is encrypted using the Secure Sockets Layer protocol (SSL). Furthermore, in a distributed monitoring infrastructure that promotes sharing of network packets and statistics between different parties, sensitive data should be *anonymized* before made publicly available, due to security, privacy, and business competition concerns that may arise between the collaborating parties.

From this picture emerges the role of the monitoring database as a kind of certification authority, which is also used as a repository of public keys used by the actors of the monitoring activity: the publication engine, the monitoring tools, and the theodolite services. Its distributed implementation is challenging, yet tightly bound to the scalability of the monitoring infrastructure.

4 Conclusions

This is a preliminary study of the issues behind the integration of passive and active techniques in a domain-oriented monitoring system. We conclude that the two techniques are complementary for the coverage of network measurements, and a domain-oriented approach is beneficial for the scalability issues that are

typical of each technique. In fact, such an approach reduces network load for active tools, and helps an efficient classification of the traffic captured by passive ones.

References

1. *LOBSTER: Large-scale Monitoring of Broadband Internet Infrastructures*. Information available at: http://www.ist-lobster.org.
2. S. Andreozzi, N. De Bortoli, S. Fantinel, A. Ghiselli, G. Tortone, and V. Cristina. Gridice: a monitoring service for the grid. In *Third Cracow Grid Workshop*, Cracow, Poland, October 2003.
3. R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor, B. Tierney, and R. Wolski. A grid monitoring architecture. Recommendation GWD-I (Rev. 16, jan. 2002), Global Grid Forum, 2000.
4. A. Ciuffoletti. The wandering token: Congestion avoidance of a shared resource. Technical Report TR-05-13, Universita' di Pisa, Largo Pontecorvo - Pisa -ITALY, May 2005.
5. A. Ciuffoletti, T. Ferrari, A. Ghiselli, and C. Vistoli. Architecture of monitoring elements for the network element modeling in a grid infrastructure. In *Proc. of Workskop on Computing in High Energy and Nuclear Physics*, La Jolla (California), March 2003.
6. Y. Fu, L. Cherkasova, W. Tang, and A. Vahdat. EtE: Passive end-to-end Internet service performance monitoring. In *Proceedings of the USENIX Annual Technical Conference*, pages 115–130, 2002.
7. L. Michael and G. Lior. The effect of packet reordering in a backbone link on application throughput. *Network, IEEE*, 16(5):28–36, 2002.
8. M. Polychronakis, K. G. Anagnostakis, E. P. Markatos, and A. Øslebø. Design of an Application Programming Interface for IP Network Monitoring. In *Proceedings of the 9th IFIP/IEEE Network Operations and Management Symposium (NOMS'04)*, pages 483–496, Apr. 2004.
9. The Globus Toolkit 4.0 Documentation. *GT Information Services: Monitoring & Discovery System (MDS)*. Available at: http://www.globus.org/toolkit/mds/.

Grid Checkpointing Architecture - a revised proposal.

R. Januszewski[1], G. Jankowski[1], J. Kovacs[2], N. Meyer[1], and R. Mikolajczak[1]

[1] Poznan Supercomputing and Networking Center,
61-704 Poznan, Noskowskiego 12/15, Poland
{Radoslaw.Januszewski,Gracjan.Jankowski,Norbert.Mayer,Rafal.Mikolajczak}@man.poznan.p
[2] Computer and Automation Research Institute of the Hungarian Academy of Sciences
1111 Budapest Kende u. 13-17. Hungary
smith@sztaki.hu

Abstract. Contemporary Grid environments are featured by an increasingly growing virtualization and distribution of resources. Such situations impose greater demands on load-balancing and fault-tolerant capabilities. The checkpoint-restart mechanism seems to be the most intuitive tool that can fulfill the specific requirements. However, as there is still a lack of widely available, production-grade checkpoint-restart tools, the higher level checkpoint-restart services are not well developed yet. One of the goals of the CoreGRID Network of Excellence is to define the high-level checkpoint-restart Grid Service and to locate it among other Grid Services. We aim to define both the abstract model of that service and the lower layer interface that will allow the service to cooperate with diverse existing and future checkpoint-restart tools. The paper is the first step on the road to this goal. It includes the overall sketch of the architecture of the considered service and its connection with the actual checkpoint-restart tools.

1 Introduction

Until now there have been few checkpointing systems that can do computing processes' checkpoints, for instance: psncLibCkpt[1], Altix C/R[2], Condor[3], libCkpt[4] and others. These checkpointing systems always have different capabilities and interfaces, and in most cases are specifically dependent on a particular OS and hardware platform. Therefore, checkpointing systems are not widely used and the existing ones always have some limitations which are different for different systems. One can try to employ the aforementioned checkpointing systems in the Grid environment. Unfortunately, in contrary to the visions expressed by experts within Next Generation Grid(s), European Grid Research 2005-2010[5] and Next Generation Grids 2, Requirements and Options for European Grids Research 2005-2010 and Beyond [6], such integration would impose high complexity. Then, if we intend to use the checkpointing functionality in Grids, we have to figure out an abstract *Grid Checkpoint Service* (*GCS*) that hides all the

complexity and underlying checkpointing systems. Moreover, that service has to fit into the more general architecture which will allow to bring into play the diverse existing and future checkpointing systems. A vision of such *GCS* and associated *Grid Checkpointing Architecture* (*GCA*) is presented in this paper. The first version of the *Grid Checkpoint Architecture* was described in the paper Towards the Grid Checkpointing Architecture [7]. As a result of exchanging experience with partners from SZTAKI and PSNC and feedback from the first paper, a new revised version of the architecture emerged.

2 Architecture

The architecture proposal included in this chapter is a revised version of *GCA* presented in the paper Towards the Grid Checkpointing Architecture [7] presented at the PPAM 2005 conference. The proposal defines four layers (*Broker*, *Checkpoint Grid Service*, *Translation* and *Core Service*) and three interfaces used to exchange information between services.

2.1 Architecture layers

The four layers (picture 1) represent mutual dependencies between different parts of *Grid Checkpoint Architecture*. Each layer hides all underlying services by providing a set of calls used to perform certain actions. The interaction is allowed only between any services placed on adjoined layers. A brief description of these layers is the following:

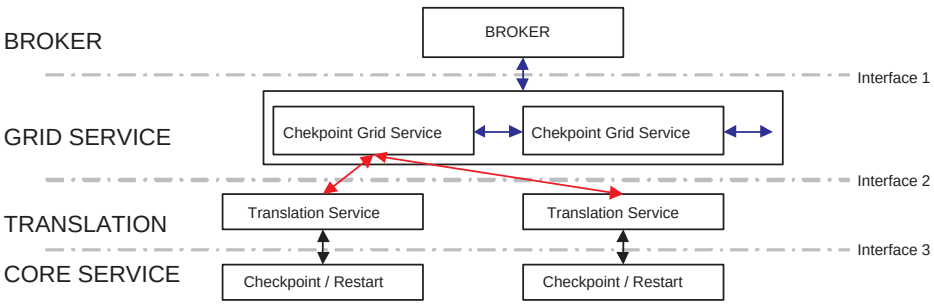


Fig. 1. Grid Checkpointing Service architecture

The GRID BROKER layer represents the Grid job manager. The first task of this layer is triggering checkpoint and restart of the applications. The decision whether to checkpoint or restart an application is made on the basis of information from monitoring services (fault detection) ,the scheduling algorithm or resource management policy. The next task of this layer is to adjust the job

description of application submitted by the user in order to ensure that the application will be checkpointable. This task may require exchange of messages with the *Grid Checkpoint Service* (*GCS*).

The GRID SERVICE layer represents a set of *Grid Checkpoint Services* (*GCS*). This layer may consist of many independent instances. Each instance is able to forward any request it cannot handle itself (peer-to-peer architecture). A *Broker* sees the set of *GCS* instances as a single service with one access point. A single *GCS* instance may have access to many *Translation Services* from the *Translation* layer (relationship one to many) Any service from this layer provides the Broker layer with all checkpointing specific functionality. Service placed on this layer manages all metadata related to images of the checkpointed applications (e.g. registers image files in the Grid Storage Service, handles the connections between applications, images and *Translation Services* etc.). The services from this layer are executing orders passed from the *Broker* layer using services from the *Translation* layer. One of the most important tasks of this service is choosing an appropriate *Translation Service* to execute the task ordered by the upper layer. An example of such task may be finding an *Translation Service*/checkpointer that will be able to handle the application, considering the requirements included in the job description. Accuracy of the *checkpointer* selection depends on how detailed the description of job requirements (regarding the desired functionality of the checkpointer) provided by the user or Grid is. The exact policy of the checkpointer selection may influence the chance of correct job checkpoint/restart.

The TRANSLATION layer represents a set of *Translation Services* (*TS*). The *TS* acts as a mediator between *GCS* and actual checkpointers. The *TS* accepts *Interface 2* messages and translates them to a format acceptable by native checkpointers (instances of the *Core Service* layer)(*Interface 3*). The *TS* instances are tightly connected with the corresponding checkpointers. For each *Core Service* checkpointer there should exist at least one *Translation Service*. The *TS* maintains information about functionality, requirements and calling semantics of the managed checkpointer (the *Core Service* layer instance). This information is used to execute checkpoint and restart operations and match the application with the checkpointer (before the application is submitted). The last function of this layer is reporting all checkpoints performed by the underlying checkpointer, even if the operation was not triggered by the *GCS*,

The CORE SERVICE layer represents the real tools used to save images of application state. Services placed on this layer will be called *checkpointers* in this paper. In general there is no assumption on what type of checkpoint it should be, it may be kernel,user or application level checkpointing. The *Core Service* may also represent some other software that is able to trigger checkpoints on a given Computing Resource (CR) (in such case the corresponding *TS* can

be considered as an interface to interface). A good example may be a local scheduler like the Sun Grid Engine that is capable of issuing checkpoint/restart commands using some third party checkpointers. This may lead to the situation when there is more than one path of access to the checkpointer. On a single computation resource there can be one or more *Translation Services*. In order to avoid problems with the *TS* selection, while configuring the *TS* on a computing resource, the administrator should indicate which is the preferred *TS* for each checkpointer.

3 Intercommunication Interface

A definition of the messages used to pass information between layers is one of the most important parts in the architecture design. In the *GCA* there are three singled out sets of messages called *interface 1*, *interface 2*, *interface 3*.

3.1 Interface 1

This interface defines communication messages exchanged between the *Broker* and the *GCS*. The meaning of messages are:

prepare_job The message is sent by the *Broker* when it wants the *GCS* to provide information about the checkpointer that is able to handle the given type of application. Depending on the original job description, the *GCS* finds the checkpointer able to create an image of the job. It may be necessary to modify the job description in order to make it work with the selected checkpointer.

checkpoint_job This message is sent by the *Broker* to the *GCS* in order to save the state image of a job. The *Broker* has to provide information necessary to identify the job. The message may contain some additional parameters: for example, a suggested checkpointer or special parameters passed to the checkpointer. The *GCS* replies with a report including information on where the image was stored, which checkpointer was used, etc.

resume_job This message initiates the process of restarting the job from the previously saved image. The *GCS* replies with a description of the job which should be submitted by the *Broker* in order to resume the given application from the saved image. [3]

[3] The *GCS* might reply with the submit job command (with the same format as users *submit_job*) however, it would require delegating the original users rights to the *GCS*. During the submit phase the Broker acts on behalf of the user anyway, so it is better to let it do all the job.

3.2 Interface 2

The *Interface 2* defines a set of messages exchanged between the *GCS* and the *Translation Service*.

prepare_submit_job The message is sent by the *GCS* after it figured out the best checkpointer. The target *TS*, basing on its knowledge about the corresponding checkpointer, has to provide a set of changes in a job description (it might be necessary to add some additional parameters) required by the checkpointer. In some cases (e.g. kernel level checkpointer) there may be no required changes.

checkpoint_job The message is used by the *GCS* to inform the *TS* that it has to call the corresponding checkpointer and make a checkpoint of the selected application. After receiving this message, the *TS* has to communicate with the checkpointer (directly or using any software that manages access to this checkpointer) in order to issue the checkpoint command (*Interface 3*). The *TS* should reply with a message consisting of the status of the operation and information about the image.

prepare_resume_job The message is sent by the *GCS* to the *TS* whem the *GCS* receives a *resume_job message* (see chapter 5.1). After the *GCS* has figured out which checkpointer was used, it has to communicate with the corresponding *TS* (in general it is irrelevant if it is exactly the same *TS* that created the image).This message indicates that the *TS* basing on an original job description and its knowledge about the underlying checkpointer (and additional parameters specified during the checkpoint) must prepare a description of the job that will resume the application from the selected image.

checkpoint_executed This message is sent by the *TS* to *GCS* when the controlled checkpointer saved the state of an application and the checkpoint was not triggered by the *checkpoint_job* call. This functionality is required in order to support self-checkpointing applications (e.g. the checkpoint is initiated at a certain point of computing). The message must contain information about the image of the application.

3.3 Interface 3

An exact definition of *Interface 3* is not a part of the *GCA* because of a variety of possible methods of triggering functions of the checkpointer(e.g. signals, environment variables, executing shell commands etc.).It is up to the *Translation Service* implementation team to design an appropriate method of communication between the *TS* and the checkpointer.

4 Interaction with other grid services

The *GCS* also utilizes other grid services. Those services were not depicted in Picture 1 because the interface and functionality of those services are not within the scope of this paper. A short description of functionality required from other grid services is the following:

1. the Information Service is used to obtain information about jobs and to store information about the executed checkpoints. This repository should be accessible by any *GCS* instance.
2. the *Storage Service* is utilized to register and store images of the applications. The detailed storage policy (replication, migration etc.) is not in the scope of the *GCA*. Transfer of the images (during the migration or restart) is performed by the *Broker* as a part of the job submission routine (when the broker prepares an environment for a job, it must ensure that files used by this application are accessible; the job images are considered to be files required by the job).
3. The *GCS* has to use Authorization Services in order to access files or manage access to information about the stored images.

5 Scenarios

This chapter will describe the behavior of services during the operations that involve the use of *GCS* submission, checkpoint, and restart of job. The migration scenario was omitted because its simplicity it may be considered as a sequence of *job_checkpoint* and *job_restart* commands.

5.1 Job submission

The job submission is a basic functionality of every broker. The submit job scenario is performed each time the user submits an application that should run in an environment managed by the *Broker*.

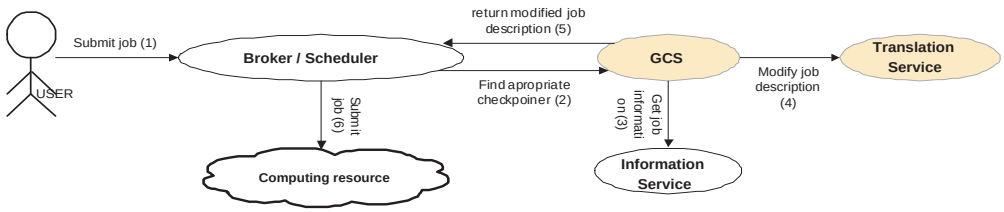


Fig. 2. Grid job submission

Job submission (with *GCS* involvement) The *GCA* extends the standard job submission scenario (the user issues the submit command, the *Broker* finds

an appropriate node and runs the job) by inserting an additional prepare stage (steps from 2 to 5 on Picture 2). This phase occurs after the *Broker* receives the description of the job from the user and before it finds the destination node. This stage requires more changes in brokers, therefore it is optional; however, it increases the chance of running the job on nodes where an appropriate checkpointer is installed. The *Broker* asks the *GCS* to modify the job description before it searches for the appropriate computing resource. This operation is performed to ensure that the job will be checkpointable. This may require adding some special options passed to the executable, adding some environment variables, specifying the need of some tool on the destination node or other changes in the job description. The whole scenario is depicted in Picture 2. A detailed description of all steps is as follows:

1. the user submits a job description to the broker (the job description can also specify the desired checkpointer along with other requirements),
2. the *Broker* forwards the job description to the *GCS* in order to obtain checkpointer-related information [*prepare_job interface 1*],
3. the *GCS* may need some additional information about the job so it may have to connect to the *Information Service* to access that information. At this stage the *GCS* must identify a checkpointer (or a set of checkpointers) and choose the most appropriate one that should be able to handle the application,
4. the *GCS* contacts a *Translation Service* responsible for the communication with the selected checkpointer. The *Translation Service* should be able to provide information on how to modify (or modify on its own) the job description (e.g. by adding some parameters, setting environment variables, adding some requirements on services installed on the computing resource, special queues definition for the local broker etc.) to make it checkpointable with the selected tool [*prepare_submit_job interface 2*],
5. after modification of a job description the *GCS* returns to the *Broker* the modified job description,
6. the *Broker* tries to find a suitable computing resource that fulfills all the requirements and submits the job to the computing resources execution module.

Job submission (without *GCS* involvement) If the *Broker* is not prepared to execute the preparation stage, the workflow is similar to the one presented in the previous scenario with the exception that steps from 2 to 5 are omitted. This scenario adhere to the existing brokers which in most cases are capable of executing checkpoint and restart commands in some way; however, they are not fully integrated with the *GCA*.

5.2 Job checkpoint

During the design phase we considered two possible scenarios of checkpoint. The first one was a standard scenario when the checkpoint is triggered by the *Broker*

(the *Broker* wants the job to be migrated or because of the selected checkpointing policy). When the checkpoint image may be created without the *GCA* involvement (e.g. because the image is created at a fixed point of computation), the second variant of checkpoint is considered.

***Broker* issued checkpoint** This is the preferred scenario of doing the checkpoint. Workflow for this case is depicted in Picture 3.

1. the *Broker* issues a checkpoint command [*checkpoint_job interface 1*]
2. the *GCS* finds the *CR* where the application is being executed, looks into the job description for the selected checkpointer and contacts with the appropriate *TS* that manages access to this checkpointer (if there is no description, the *GCS* has to find the best checkpointer from those installed on the *CR*) The [*checkpoint_job interface 2*] command is sent to the selected *TS*,
3. the *TS* is executing the checkpoint command calling the underlying checkpointer,
4. the checkpointer reports the status of the checkpoint operation to the *TS*,
5. the *TS* returns the information about the checkpoint image and status of the operation
6. the *TS* registers information about the image in the *Information Service*. This information will be used during the restart phase,
7. the files containing the checkpoint image are registered in the *Storage Service* to make them accessible by other Grid Services,
8. the status of the whole checkpoint operation is returned to the *Broker*.

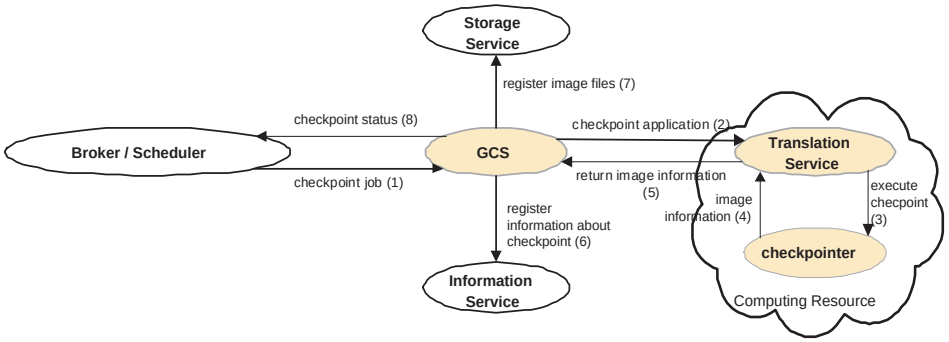


Fig. 3. Checkpoint issued by the broker

Independent checkpoint This is the second variant of a checkpoint scenario if the checkpointer does not support "triggered" checkpoint (by the *Broker*) or performs checkpointing after a certain period of time or at fixed point of computation. In order to allow the application to be restarted from the image created in that way, the *TS* has to report every checkpoint that is executed by

the underlying checkpointer. The Translation Service provides the *GCS* with information about image placement, date and time of image creation (according to possibilities of obtaining this information). For each information about such independent checkpoint the *GCS* has to try to find an appropriate job issued by the *Broker*. Only if such mapping is possible, the information about a checkpoint for the given image is stored in the *IS*. The steps in this scenario are the following:

1. at some specified point of time the checkpoint is executed (the event that triggered the checkpoint is outside the *GCA*),
2. the *TS* has to intercept information about the checkpoint (4)[4],
3. information about the checkpoint is sent to the *GCS*. The *GCS* has to check if there is a Grid application with the local id equal to the one passed with the message. The type of the local id may depend on the *TS* (process id, parent process id, id of the job in the local queuing system etc.) that sends the information [*checkpoint_executed interface 2*] (5),
4. if the *GCS* can match the local id returned by the *TS* in the previous step with any grid application, the information about the image is stored in the Information Service (6),
5. files with the application image are registered in the Storage Service (7).

5.3 Job restart

In the current version of the *GCA* the restart of the application is divided into two stages. The first stage of the restart procedure is initiated by the grid broker by calling the *GCS*. During this stage a description of a special job which will be used by the *Broker* to trigger the restart is prepared. A special job description is based on the description and requirements of the original job because the node where the resumed job will run has to fulfill all the requirements of the original job. Depending on the checkpointer that was used to create the images, some additional requirements may be added or changed. The second stage is performed by the broker and is identical with the normal job submission scenario. The resume of the application is performed by the execution module of the Grid according to a description provided by the *GCS* and *TS*. The workflow during the restart of a job is depicted in Picture 5 The stages of job restart:

1. the *Broker* sends a *resume_job* [*interface 1*] message to the *GCA*. This message is sent according to some policy (perhaps after a job failure detection or part of a migration procedure),
2. the *GCS* checks if there are any images for the given job. If such image is prepared, the *GCS* has to select the most appropriate image (probably the most recently created). At the next stage the *GCS* has to find a *TS* for the checkpointer that created the image and ask it to prepare a set of changes in the original job description [*prepare_job_resume interface 2*],
3. the *TS* has to parse the job description along and specify a set of changes in the original job description that will cause the job to be resumed. The set of changes/modified job description is sent back to the *GCS*,

[4] Numbers in the brackets refers to stages depicted in Picture 3.

4. the *GCS* may make some further changes (e.g. adding the files of the image to the job description in order to cause them to be copied to the destination node) and sends a modified job description,
5. the *Broker* finds an appropriate node according to the requirements in the job description and submits the job.

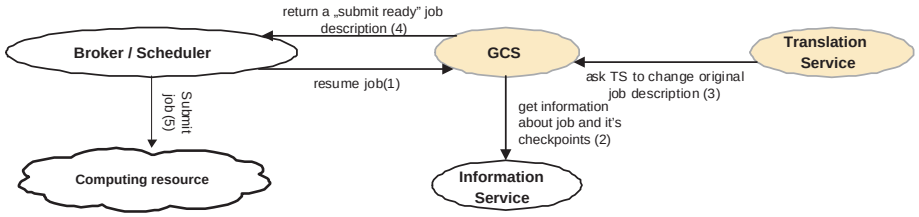


Fig. 4. Message exchange during job restart

6 Conclusions

The *GCA* in the current form is a very flexible architecture that is able to use much of the existing and future checkpoint and restart toolkits. Introduction of this service should encourage users to use the Grid computing because of providing higher fault tolerance and safety level of application which are the Achilles heel of a system consisting of thousands of distributed nodes. There are other projects focusing on similar topics such as GGF WG [8]; however, due to a lack of space their comparison with our project was omitted.

References

1. http://checkpointing.psnc.pl/Progress/psncLibCkpt/
2. Checkpoint/Restart mechanism for multiprocess applications implemented within SGIGrid Project, Gracjan Jankowski, Rafa Mikoajczak, Radosaw Januszewski, CGW2004..
3. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System, Michael Litzkow, Todd Tannenbaun, Jim Basney, and Miron Livny; Computer Sciences Department University of Wisconsin-Madison.
4. Libckpt: Transparent Checkpointing under Unix', Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January, 1995.
5. Next Generation Grid(s), European Grid Research 2005-2010, Expert Group Report, 16th June 2003.
6. Next Generation Grids 2, Requirements and Options for European Grids Research 2005-2010 and Beyond, Expert Group Report, July 2004.
7. Towards the Grid Checkpointing Architecture, G. Jankowski, J. Kovacs, R. Mikolajczak, R. Januszewski, N. Meyer Poznan Supercomputing and Networking Center
8. http://www.ggf.org/

Simulating Grid Schedulers with Deadlines and Co-Allocation

Alexis Ballier[2], Eddy Caron[2], Dick Epema[1], and Hashim Mohamed[1]

[1] Delft University of Technology, Delft, the Netherlands
[2] LIP ENS Lyon, UMR CNRS - ENS Lyon - UCB Lyon - INRIA 5668, France

Abstract. One of the true challenges in resource management in grids is supporting co-allocation, that is, the allocation of resources in multiple autonomous subsystems of a grid to single jobs. With reservation-based local schedulers, a grid scheduler can reserve processors with these schedulers to achieve simultaneous processor availability. However, with queuing-based local schedulers, it is much more difficult to guarantee this. In this paper we present mechanisms and policies for working around the lack of reservation mechanisms for jobs with deadlines that require co-allocation, and simulations of these mechanisms and policies.

1 Introduction

Over the past years, multi-cluster systems consisting of several clusters containing a total of hundreds to thousands of CPUs connected through a wide area network (WAN) have become available. Examples of such systems are the French Grid5000 system [3] and the Dutch Distributed ASCI Supercomputer (DAS)[5]. One of the challenges in resource management in such systems is to allow the jobs access to resources (processors, memory, etc.) in multiple locations simultaneously—so-called *co-allocation*. In order to use co-allocation, users submit jobs that consist of a set of *components*, each of which has to run on a single cluster. The principle of co-allocation is that the components of a single job have to start at the same time.

Co-allocation has already been studied with simulations and has been proven to be a viable option [2, 6]. A well-known implementation of a co-allocation mechanism is DUROC [4], which is also used in the KOALA scheduler. KOALA, which is a processor and data co-allocator developed for the DAS system [8, 7], adds fault tolerance and scheduling policies to DUROC, and support for a range of job types. KOALA been released in the DAS for general use in september 2005 (`www.st.ewi.tudelft.nl/koala`).

One of the main difficulties of processor co-allocation is to have processors available in multiple clusters with autonomous schedulers at the same time. When such schedulers support (advance) reservations, a grid scheduler can try to reserve the same time slot with each of these schedulers. However, with queuing-based local schedulers such as SGE (now called SUN N1 Grid Engine) [9], which is used in the DAS, this is of course not possible. Therefore, we have designed and implemented in KOALA mechanisms and policies for placing jobs (i.e., finding suitable executions sites for jobs) and for claiming processors before jobs are supposed to start in order to guarantee processor availability at their start time.

In this paper we present a simulation study of these mechanisms and policies where we assume that jobs that require co-allocation have a *deadline* attached to them. In Section 2, we describe the model we use for the simulations, and in Section 3 we discuss and analyse the results of these simulations. Finally, in Section 4 we conclude and introduce future work.

2 The Model

In this section we describe the system and scheduling model to be used in our simulations. Our goal is to test different policies of grid schedulers with co-allocation and deadlines. With co-allocation, jobs may consist of separate components, each of which requires a certain number of processors in a single cluster. It is up to the grid scheduler to assign the components to the clusters. Deadlines allow a user to specify a precise job start time; when the job cannot be started at that time, its results will be useless or the system may just give up trying to schedule the job and leave it to the user to re-submit it.

One of the main problems of co-allocation is to ensure that all job components will be started at a specified time simultaneously. In queuing-based systems, there is no guarantee that the required processors will be free at a given time. On the other hand, busy processors may be freed on demand in order to accommodate a co-allocated (or *global*) job that has reached its deadline. Therefore, we assume that the possibility exists to kill jobs that do not require co-allocation (*local jobs*); all results of such jobs are lost, and they have to be started again at a later time. In our model, the global jobs have *deadlines* at which they should start, otherwise they will be considered as failed.

As an alternative to this model (or rather, to the interpretation of deadlines), one may consider jobs with components that have input files which first have to be moved to the locations where the components will run. When these locations have been fixed, we may estimate the file transfer times, and set the start time of a job as the time of fixing these locations plus the maximum transfer time. Then this start time can play the same role as a real deadline. The difference is that in our model, if the deadline is not met, the job fails, while in this alternative, the job may still be allowed to start, possibly at different locations.

2.1 System model

We assume a multicluster environment with C clusters, which for the sake of simplicity is considered to be homogeneous (*e.g.*, every processor has the same power). We also assume in our simulations that all clusters are of identical size, which we denote by N the number of nodes. Each cluster has a local scheduler that applies the First Come First Served (FCFS) policy to single-component local jobs sent by the local users. The scheduler can kill those local jobs if needed. When they arrive, the new jobs requiring co-allocation are sent to a single global queue called the *placement queue*, and here the jobs wait to be assigned to some clusters.

In our model we only consider unordered jobs, which means that the execution sites of a job are not specified by the user, but that the scheduler must choose them. A Poisson arrival process is used for the jobs requiring co-allocation and for the single-component

jobs for each cluster, with arrival rates λ_g for global jobs and λ_l for the local ones in each cluster.

A job consists of a number of components that have to start simultaneously. The number of components in a multi-component job is generated from the uniform distribution on $[2, C]$. The number of components can be 1 only for local jobs which do not require co-allocation. The deadline for a job (or rather the time between its submission and its required start time) is also chosen randomly with a uniform distribution on $[D_{min}, D_{max}]$, for some D_{min} and D_{max}. The number of processors needed by a component is taken from the interval $I_s = [4, S]$, where S is the size of the smallest cluster. Each component of a job will require the same number of processors. Two methods are used to generate that size. The first is the uniform distribution on I_s. The second is more realistic and we have used it in previous simulation work in order to have more sizes that are powers of two as well as more small sizes [2]. In this distribution, which we call the Realistic Synthetic Distribution, a job component has a probability of q^i/Q to be of size i if i is not a power of two, and $3q^i/Q$ to be of size i if i is a power of two. Here $0 < q < 1$, and the value of Q is chosen to make the sum of the probabilities equal to 1. The factor 3 is made to increase the probability to have a size that is a power of 2 and q^i to increase the chance to have a small size.

Finally, the computation time of the job has an exponential distribution with parameter μ_g for the global jobs and μ_l for the local jobs.

2.2 Scheduling Policies

In this section the so-called *Repeated Placement Policy* (RPP) will be described. Suppose a co-allocated job with deadline D is submitted at time S. The RPP has a parameter L_p, $0 < L_p < 1$, which is used in the following way. The first placement try will be at time PT_0, with:

$$PT_0 = S + L_p \cdot (D - S).$$

If placement does not succeed at PT_m, the next try will be at PT_{m+1}, defined as:

$$PT_{m+1} = PT_m + L_p \cdot (D - PT_m).$$

As this policy can be applied forever, a limit on m has to be set, which will be denoted M_p. If the job is not placed at PT_M, it is considered as failed. When a placement try is successful, the processors allocated to the job are claimed immediately. Note that with this policy, the global jobs are not necessarily scheduled in FCFS order.

In our simulations, the Worst Fit placement policy is used for job placement, which means that the components are placed on the clusters with the most idle processors. With this method, a sort of load balancing is performed. We assume that different components of the same job can be placed on the same cluster.

If the deadline of a newly submitted multi-component job is very far away in the future, it may be preferable to wait until a certain time before considering the job. The scheduler will simply ignore the job until $T = D - I$, where D is the deadline and I is the *Ignore* parameter of the scheduler. We will denote by Wait-X the policy with $I = X$. With set I to ∞, no job will ever be ignored. In our model, there are a single global queue for the multi-components jobs, and a local queue for each cluster for single-component jobs. In order to give global or local jobs priority, we define the following policies [1]:

GP: When a global job has reached its deadline and not sufficient processors are idle, local jobs are killed.

LP: When we cannot claim sufficient numbers of processors for a global multi-component job, that job fails.

2.3 Performance metrics

In order to assess the performances of the different scheduling policies, we will use the following metrics:

The global job success rate: The percentage of co-allocated jobs that were started successfully at their deadline.

The local job kill rate: The percentage of local jobs that have been killed.

The total load: The average percentage of busy processors over the entire system.

The global load: The percentage of the total computing power that is used for computing the global jobs. It represents the effective computing power that the scheduler has been able to get from the grid.

The processor wasted time: The percentage of the total computing power that is wasted because of claiming processors before the actual deadlines of jobs.

3 Simulations

In this section we present our simulation results. We first discuss the parameters of the simulation, then we simulate the *pure* Repeated Placing Policy, which does not ignore jobs, and finally discuss the Wait-X policies.

3.1 Setting the parameters

All our simulations are for a multicluster system consisting of 4 clusters of 32 processors each, and unless specified otherwise, the GP policy is used. The number of processors needed by a local job (its size) is denoted S_l and is generated on the interval $[1; 32]$ with the Realistic Synthetic Distribution with parameter $q = 0.9$. The expected value of S_l is $E[S_l] = 6.95$.

We set $\mu_l = 0.01$ so that local jobs have an average runtime of 100 seconds. Then the (requested) local load U_l in every cluster due to local jobs is equal to:

$$U_l = \frac{\lambda_l \cdot E[S_l]}{\mu_l \cdot N}.$$

We run simulations with a *low local load* of 30% and a *high local load* of 60%, which are reasonable values on our DAS system, which is a research system rather than a production facility.

We denote by S_g the size of a component of a global job, which is taken on the interval $[4; 32]$ and which is also generated using the Realistic Synthetic Distribution with parameter $q = 0.9$. The expected value of S_g is $E[S_g] = 10.44$. The number of components of a global job, N_c, is taken uniformally on the interval $[2; 4]$. In our

simulations we set $\mu_g = 0.005$, so the global jobs have an average runtime of 200 seconds. The (requested) global load, denoted U_g, is given by:

$$U_g = \frac{\lambda_g \cdot E[N_c] \cdot E[S_g]}{\mu_g \cdot N \cdot C}.$$

It should be noted that we cannot compute the actual (local or global) load in the system. The reasons are that local jobs may be killed, there is processor wasted time because we claim processors early, and global jobs may fail because they don't meet their deadlines. However, the useful load can be computed.

We run simulations with a *low global load* of 20% and a a *high global load* of 40%. The results of a first general simulation, with $L_p = 0.7$, are presented in Figure 1 to show the general behavior of the simulated system. We can check that these results are well correlated with the computed values.

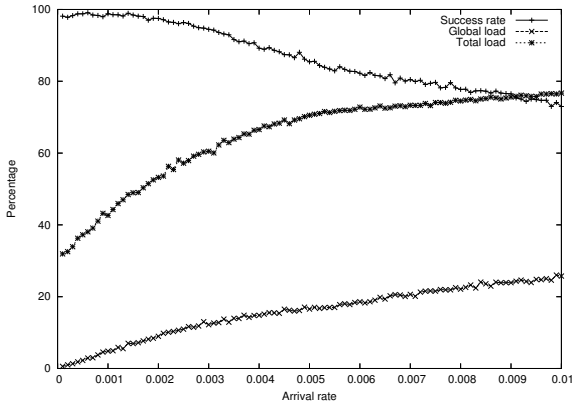


Fig. 1. Some metrics as a function of the arrival rate of global jobs with a low local load (30%).

3.2 The Pure Repeated Placing Policy

In this section we study the influence of the parameters of the Repeated Placing Policy, which is nothing else than the Wait-∞ policy. The deadline is chosen uniformally on the interval $[1; 3599]$. The parameter we vary is L_p.

We first study the Wait-∞ policy with a low local load. We expected that the success rate of global jobs will be higher for lower values of L_p, but this is not the case, as shown in Figure 2 for a low local load. These results may seem strange because RPP is designed to have a high success rate for global jobs. The processors for the global jobs are claimed earlier in order to ensure that their availability at the deadlines. In fact, the first jobs have indeed a greater success rate but the ones that come after them find fewer free processors, what causes them to fail. This analysis is clear when analysing the total load as a function of L_p. It seems preferable to set L_p to 1 with any arrival rate of the global jobs, at least for a low local load. However, the conclusion may be different with a high local load.

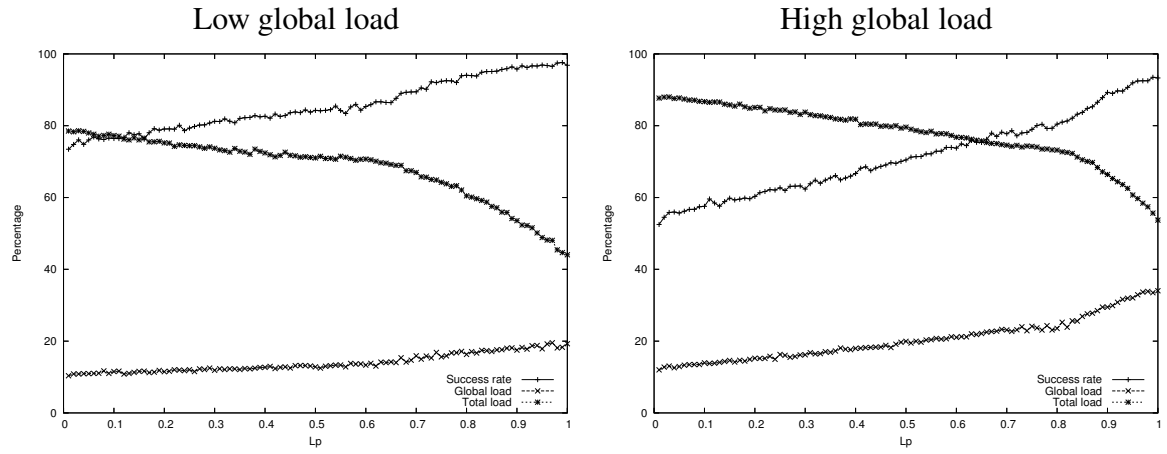


Fig. 2. The influence of L_p with a low local load.

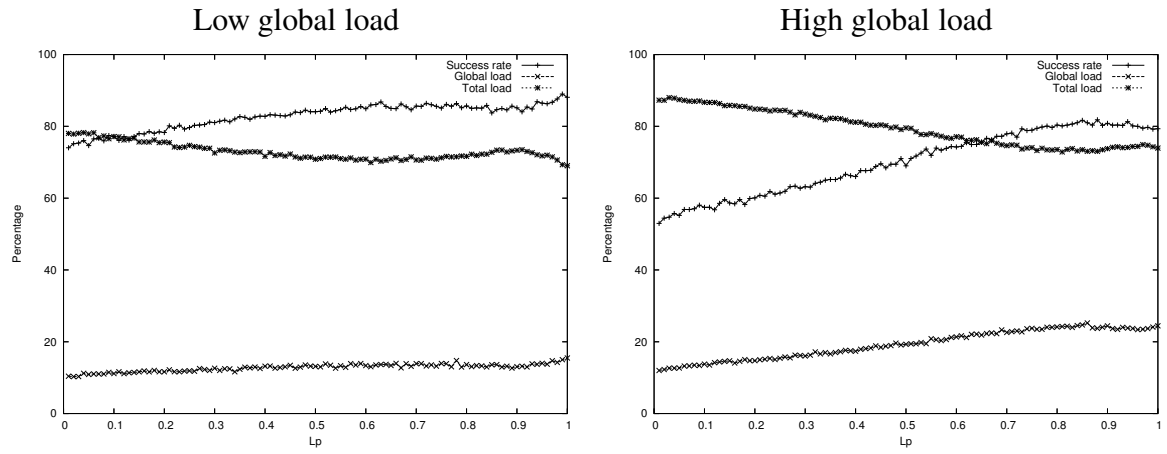


Fig. 3. The influence of L_p with a high local load.

Therefore, we study the influence of L_p with a high local load (60%). The results of those simulations shown in Figures 3 are very similar to the ones with a low local load. The success rate decreases a little bit when L_p is close to 1 when both the local and global loads are high, which is due to the fact that the total requested load is equal to 100%. This leads to the conclusion that the best way to meet the deadlines is simply to try to run the jobs at their starting deadlines with the hope that there will be enough free processors.

3.3 The Wait-X Policies

We have shown that the pure RPP (Wait-∞) is not efficient since a value of L_p close to 1 is the best setting, which causes much processor time to be wasted. However, as described in Section 2.2, it might be preferable to simply ignore jobs until I seconds before their starting deadline. Since the scheduling policy may affect the results, we study both LP and GP policies. In this section we fix L_p at 0.7.

According to the results shown in Figure 4, ignoring the jobs until 100 seconds or less before their starting deadline gives more or less the same results, while a pure RPP and ignoring until 1000 seconds before the deadline gives less good results. The Wait-0,

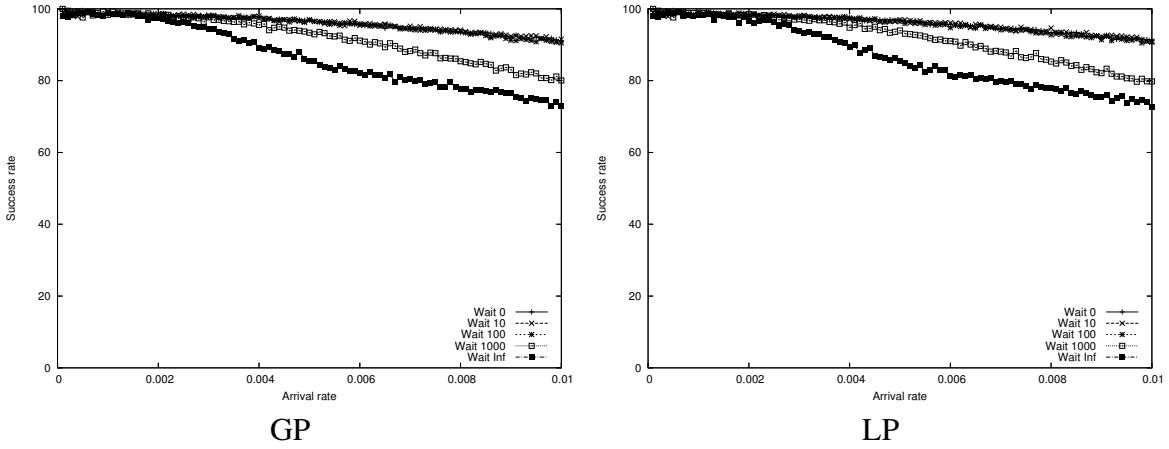


Fig. 4. Comparison of different ignoring policies with a low local load varying the global arrival rate λ_g.

Wait-10, and Wait-100 policies seem to be the most suitable choices for any arrival rate of the global jobs. We will prefer the Wait-10 policy to the two others because we want to have the possibility to place a job again in the case of failure, what we cannot do with the Wait-0 policy. We also do not want to have the overhead of applying the RPP over a too large interval of time.

The next parameter we investigated the influence of is the deadline (that is, the time between submission and required start time) of global jobs. Since the Wait-10 policy was concluded to be the most suitable scheduling policy we compare it to the pure RPP. We compare the success rates of the global jobs depending on their deadline. The results are in Figures 5 and 6. As expected, in both cases, the behavior of the Wait-10 scheduling policy is not affected by the value of the deadline, and the Wait-10 policy has better results than the pure RPP policy for any global load.

We now study the influence of the load due to the single-component local jobs on both the RPP and the Wait-10 policies. The GP scheduling policy is used because, with high local loads, the global jobs may not be able to run with the LP policy. As shown in Figure 7, the Wait-10 and pure RPP policies have rather the same success rate while varying the local load.

Finally, we trace the impact of the different scheduling policies on the local jobs. The policy will always be GP because a LP policy may not influence the local jobs. As we can see in Figure 8, the pure RPP does not let the local jobs run properly while the other policies are much nicer with them, with also better results for the global jobs as we have shown previously. The results shown in Figure 9 are also in favor of the Wait-10 policy because when I is set to great values such as 1000 or ∞ there is a considerable amount of local jobs that are killed.

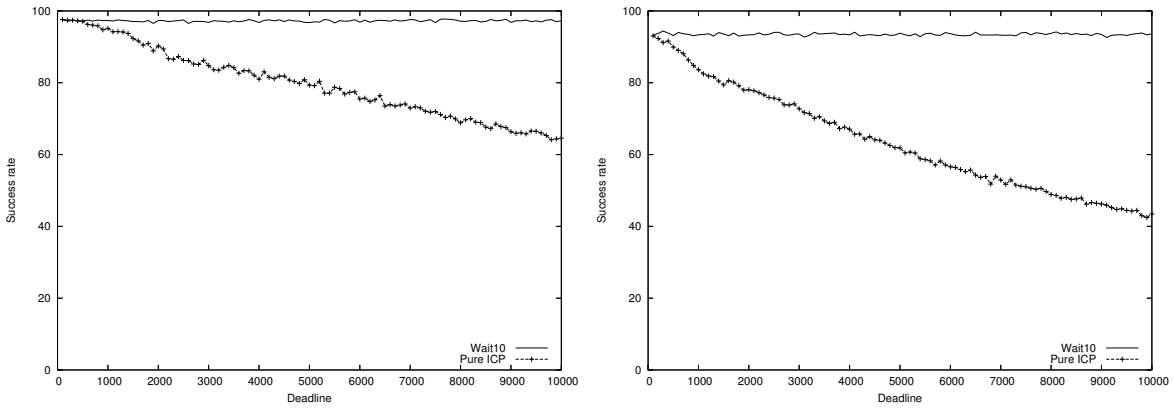


Fig. 5. The influence of the deadline with a low local load.

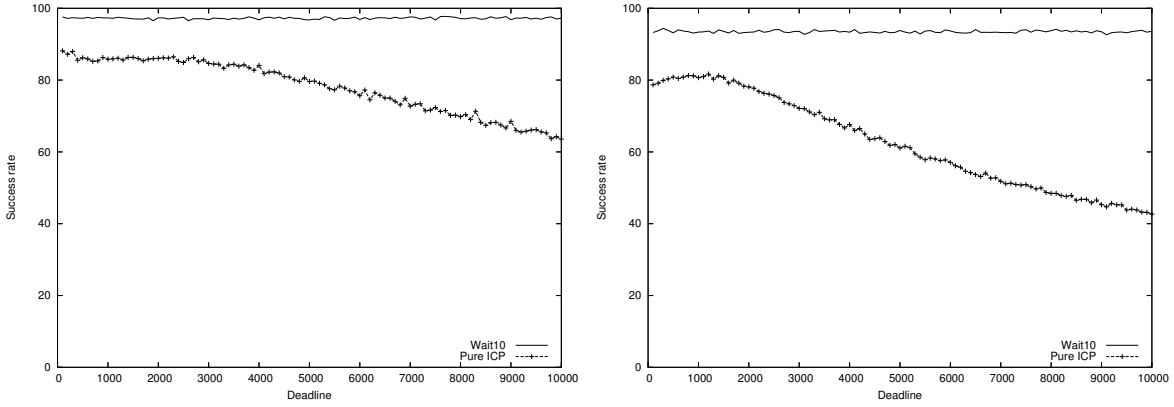


Fig. 6. The influence of the deadline with a high local load.

4 Conclusions

In this paper we have presented a simulation study of grid schedulers with deadlines and co-allocation based on queuing-based local schedulers. We have shown that it is better to try scheduling global jobs a short period of time before their deadlines. Considering the jobs too early may cause many jobs to fail; the first jobs, indeed, run fine but waste a lot of processor time, while the next ones do not have enough processors to run.

A first extension to this work could be to consider the communication overheads that happen in real grids. The Wait-10 policy, which was found to be the best policy, may not be that good and the best value for the Ignore parameter I may depend on these overheads. Second, we may also extend this work by considering the parameter I as a priority parameter. As a final extension, we may develop and test policies which try to schedule global jobs more aggressively when the local loads are much higher, as they are in production installations.

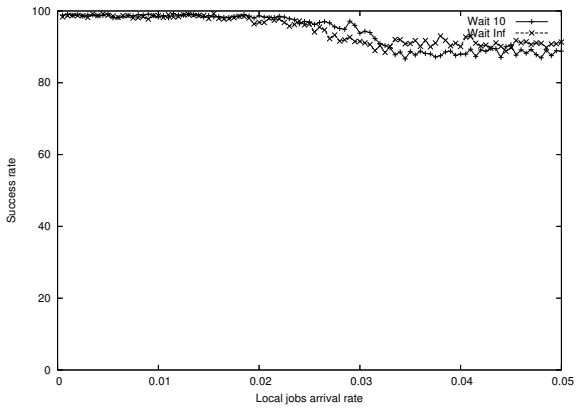


Fig. 7. The influence of the local load on the Wait-10 and pure RPP policies with a low global load.

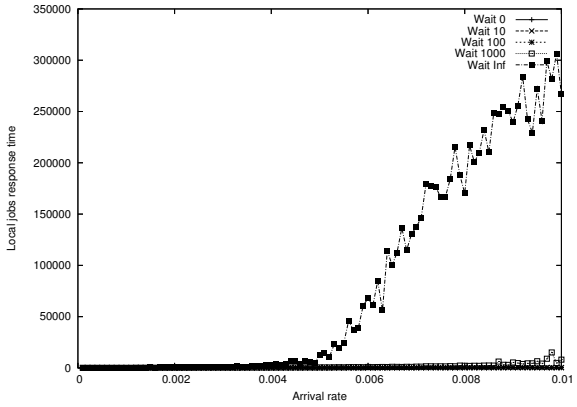


Fig. 8. The response time of local jobs as a function of the arrival rate of global jobs with different scheduling policies with a low local load.

Acknowledgments

This work was carried out in the context of the Virtual Laboratory for e-Science project (`www.vl-e.nl`), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ). In addition, this research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

References

1. A.I.D. Bucur and D.H.J. Epema. Priorities among Multiple Queues for Processor Co-Allocation in Multicluster Systems. In *Proc. of the 36th Annual Simulation Symp.*, pages 15–27. IEEE Computer Society Press, 2003.
2. A.I.D. Bucur and D.H.J. Epema. The Performance of Processor Co-Allocation in Multicluster Systems. In *Proc. of the 3rd IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2003)*, pages 302–309. IEEE Computer Society Press, 2003.

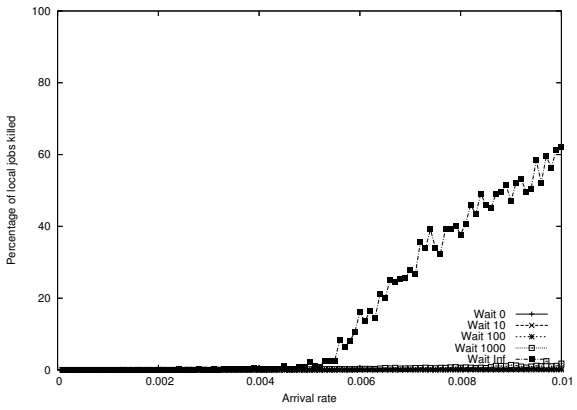


Fig. 9. The percentage of local jobs killed as a function of the global jobs arrival rate with different scheduling policies with a low local load.

3. F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *Grid'2005 Workshop*, Seattle, USA, November 13-14 2005. IEEE/ACM.
4. K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *Proc. of the 8th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.
5. The Distributed ASCI Supercomputer (DAS). `www.cs.vu.nl/das2`.
6. C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On Advantages of Grid Computing for Parallel Job Scheduling. In *Proc. of the 2nd IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2002)*, pages 39–46, 2002.
7. Hashim H. Mohamed and Dick H. J. Epema. The design and implementation of the KOALA co-allocating grid scheduler. In *European Grid Conference*, pages 640–650, 2005.
8. H.H. Mohamed and D.H.J. Epema. Experiences with the KOALA Co-Allocating Scheduler in Multiclusters. In *Proc. of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005)*, 2005 (to appear, see `www.pds.ewi.tudelft.nl/~epema/publications.html`).
9. The Sun Grid Engine. `http://gridengine.sunsource.net`.

Towards a scheduling policy for hybrid methods on computational Grids

Pierre Manneback[1], Guy Bergère[2], Nahid Emad[3], Ralf Gruber[4], Vincent Keller[4], Pierre Kuonen[5], Tuan Anh Nguyen[5], Sébastien Noël[1], and Serge Petiton[2]

[1] Faculté Polytechnique de Mons and CETIC, Mons, Belgium
{Pierre.Manneback,Sebastien.Noel}@fpms.ac.be
[2] INRIA-Futurs, LIFL, USTL, Villeneuve d'Ascq, France
{Bergere,Petiton}@lifl.fr
[3] Laboratoire PRISM, UVSQ, Versailles, France
nahid.emad@prism.uvsq.fr
[4] Département STI-SGM, EPFL, Lausanne, Switzerland
{Ralf.Gruber,Vincent.Keller}@epfl.ch
[5] University of Applied Sciences of Fribourg, Fribourg, Switzerland
{Tuan.Nguyen,Pierre.Kuonen}@eif.ch

Abstract. In this paper, we propose a cost model for running particular component based applications on a computational Grid. This cost is evaluated by a metascheduler and negotiated with the user by a broker. A specific set of applications is considered: hybrid methods, where components have to be launched simultaneously. [6]

1 Introduction

Hybrid methods mix together several different iterative methods or several copies of the same method in order to solve efficiently some numerical problems. They can be considered as alternative to classical methods if two properties are matched: the convergence of the hybrid method has to be faster than each individual method and merging cost between methods has to be low in comparison with the convergence speed-up.

Hybrid methods are used in different fields such as combinatorial optimization [7], numerical linear algebra [5, 3] and general asynchronous iterative schemes [1]. They are well suited for large parallel heterogeneous environments such as Grids, since every method can run asynchronously at each own pace. In order to accelerate convergence, they need however regular interactions, and therefore a suitable coschedule has to be proposed. In this paper, we introduce the coscheduling problem for a specific class of hybrid methods. We start in the next

[6] This research work is carried out under the FP6 Network Of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265). It is a collaborative work between several partners of Resource Management and Scheduling Virtual Institute (WP6).

section by describing a proposal for a cost model. We pursue in section 3 by describing a class of hybrid methods (hybrid iterative methods for linear algebra) as a case study for the scheduling. We continue in section 4 by presenting POP-C++, which is a programming environment easying the development of parallel applications on the Grid, and is well suited to deploy hybrid methods.

2 Description of a cost model

Computational grids offer a considerable set of resources to run HPC applications. Resource management and scheduling are of paramount importance to exploit economically these grids. We have to avoid for instance to run non adapted applications on some resources and spoiled them.

Let consider one parallel application A composed of C_1, C_2, ..., C_n components (i.e. parallel tasks), which interact together (inter-parallelism). Components have an internal parallel structure (intra-parallelism) and can be composed and described by a workflow [2]. A computational grid is composed of R_1,...,R_r resources, each of them disposing of a local resource information system.

We assume that one component C_k can only be placed on a certain amount of nodes on one or more resources R_i (each composed of p_i nodes). Each resource can run one or more components. A node is composed of one or a few processors. We denote P_{ij} the node j on a resource i. At any time, each node can only be devoted to at most one component. We will suppose also that the multiprocessor resources have a distributed-memory architecture. The Grid architecture we focus on is a dedicated computational Grid, composed of several clusters.

A schedule S will be denoted by a list of mappings

$$C_k \rightarrow (\{P_{ij}\}_k, t_k^{start}, t_k^{end}) \qquad\qquad k = 1, \ldots, n \tag{1}$$

t_k^{start} and t_k^{end} are respectively the starting time and the estimated ending time of a component C_k on the set of nodes $\{P_{ij}\}_k$.

The workflow is defined by two types of constraints:

- a partial order precedence relation $\prec$, $C_{k_1} \prec C_{k_2}$ meaning that $t_{k_1}^{end} < t_{k_2}^{start}$. We denote by $\mathcal{P}$ the set of all couples (k_1, k_2) such that $C_{k_1} \prec C_{k_2}$. These constraints have to be strictly respected.
- a simultaneity relation $\simeq$, $C_{k_1} \simeq C_{k_2}$ meaning that $t_{k_1}^{start}$ and $t_{k_2}^{start}$ should be equal. We denote by $\mathcal{S}$ the set of all couples (k_1, k_2) such that $C_{k_1} \simeq C_{k_2}$.

This last set of constraints is very important for hybrid methods where different collaborative components should be launched at the same time.

The basic model for scheduling the components C_1, C_2,..., C_n on the grid for one HPC application A is defined as:

Find a schedule S such that it minimizes cost(A,S) with respect to constraints:

$$\forall P_{ij} \in \{P_{ij}\}_k, P_{ij} \textit{ is admissible for running } C_k \tag{2}$$

$$t^{end}(A,S) \leq t^{end}_{max}(A) \tag{3}$$

$$cost(A,S) \leq cost_{max}(A) \tag{4}$$

$$C_{k_1} \prec C_{k_2} \qquad \forall (k_1,k_2) \in \mathcal{P} \tag{5}$$

$$C_{k_1} \simeq C_{k_2} \qquad \forall (k_1,k_2) \in \mathcal{S} \tag{6}$$

The function $cost(A,S)$, which represents the cost for the user, has to take account of different parameters: cpu time, elapsed time, communication volume, storage cost, number of used processors on a resource R_i, usage cost of this resource, execution time interval, power consumption, etc. It is evaluated by the metascheduler, on the basis of the information provided by the local schedulers.

It will be the task of a resource broker to propose a suitable allocation and schedule. This broker will invoke a metascheduler, which will call the local resource information system on each resource or pool of resources. Each local scheduler will reply by a service message describing availabilities, nodes specifities (e.g. softwares and libraries) and reservation costs (cost per hour for each type of nodes, cost for a certain volume of transferred data, cost for power consumption, etc.). The metascheduler, by the mean of the data repository, will be able to select suitable schedules that will meet users and resource administrators requirements. We intend to exploit the UniCORE/MetaScheduler/ISS Grid middleware [6]. While this approach is feasible for small sets of resources, it would not scale up to large scale Grids, where a discovery and preselection phase would have to be implemented.

The admissibility of the allocation of A (2) lies in that all nodes in $\{P_{ij}\}_k$ have to meet all the requirements of C_k in terms of permissions, operating system, software, licenses, storage, memory, minimal and maximal number of processors and local policy.

The end user will give his requirements by specifying two parameters: the maximal cost $cost_{max}(A)$ that he wants to pay for running his application and the deadline upper limit $t^{end}_{max}(A)$. The metascheduler will propose suitable resources for each component C_k, with table of costs, starting time and ending time. If both user requirements $cost_{max}(A)$ and $t^{end}_{max}(A)$ can not be simultaneously met, schedule bids will be proposed in two groups: the first one with schedules respecting the cost limit; the latter one with schedules respecting the ending time limit. We will not consider here the problem of rescheduling components or preemption of resources.

In order to illustrate the cost model, let us consider an application with two components C_1 and C_2 in a serial workflow and a grid made of 3 resources R_1, R_2, R_3 composed of 16, 4, and 16 computing nodes, respectively. The collected information about resources (number of processors available during a certain time interval, available librairies and cost) is presented in Table 1. In this example, the resource R_3 is the most expensive one. The cost is defined by each resource administrator and the high cost of a resource will generally means a high performance network and high performance nodes. An administrator can

resource	#proc	t^{start}	t^{end}	supplied libraries	cost / (t.u. × proc)
R_1	12	1	20	L_2	20
R_2	4	1	6	L_1, L_2, L_3	20
R_2	4	7	20	L_1, L_2, L_3	15
R_3	8	5	15	L_2, L_3	25

Table 1. Collected information from each local scheduler by the metascheduler

impose high cost without proposing high performance resources to keep the resource unused (e.g. for local usage). R_3 is assumed to be perfectly scalable and its per processor computing time is 25% lower than R_2. On resource R_2, scalability is linear until 2 processors, and has a value of 3.2 on 4 processors. Each local scheduler can impose varying costs depending on specified time intervals. For instance, the R_2 administrator encourages the use of R_2 after time 6 by applying attractive costs. User requirements are identified by cost and completion time bounds. In this example, user has fixed $cost_{max}(A)$ at 540 units and $t^{end}_{max}(A)$ at 10 time units. The user do not give any information concerning the number of required processors : this kind of information will be provided by the Gamma model of the ISS [4]. We consider that C_1 needs the library L_1 and C_2 needs libraries L_2 and L_3. Therefore, C_1 is admissible on resource R_2 only and C_2 on resources R_2 and R_3. We suppose that the processor time for C_1 on R_2 is 8 time units, independent of the number of processors used, and the processing time of C_2 is 16 units on R_3, thus 20 units on a 2 processor R_2, and 25 units for a 4 processor R_2. Such information can be obtained through the Gamma model. The resource broker will gather from the metascheduler and the local schedulers potential schedules of the type illustrated at Table 2.

#	comp	resource	#proc	#start	#end	cost
1	C_1	R_2	2	1	4	160
2	C_1	R_2	2	5	8	140
3	C_1	R_2	2	7	11	120
4	C_1	R_2	4	7	8	120
5	C_1	R_2	4	1	2	160
6	C_2	R_2	2	5	14	320
7	C_2	R_2	2	9	18	300
8	C_2	R_2	4	3	$8\frac{1}{4}$	455
9	C_2	R_2	4	7	$12\frac{1}{4}$	375
10	C_2	R_3	8	5	6	400
11	C_2	R_3	8	9	10	400

Table 2. Scheduling of components on available and admissible resources.

Taking into account the precedence constraint ($C_1 \prec C_2$), some bids can be proposed for which:

1. $cost_{max}(A)$ is respected
2. $t^{end}_{max}(A)$ is respected
3. Both criterions are respected

Therefore, the metascheduler will propose three bids as shown in Table 3, all of them respecting the sequential workflow. The first one is of minimal cost of 420 cost units, but lasts 18 time units instead of 10, as requested by the user. The second one has a minimal ending time of $t^{end} = 6$, but costs 560 units instead of 540 demanded by the user. The last one respects both constraints.

#	sched	t^{end}	cost
$bid1$	$4 \rightarrow 7$	18	420
$bid2$	$5 \rightarrow 10$	6	560
$bid3$	$4 \rightarrow 11$	10	520

Table 3. Scheduling bids proposed by the broker. The notation i $\rightarrow$ j means that scheduling is based on rows i and j of Table 2.

Our proposed allocation and scheduling problem is combinatorial. Some heuristics will have to be exploited in order to explore the set of admissible schedules and propose consistent bids. The idea is to develop a *Contract Manager* between the user and the Grid. Different bids can be proposed to the user, with different costs respecting the user requirements. The plausibility of the given ending time should be estimated in such a way that realistic contract offers can be proposed. Therefore, an evaluation phase can be required in order to evaluate the size of the user application A (computation and communication requirements). The usage of a data repository as proposed in the Intelligent Scheduling System [6] will be necessary for this phase.

One major difficulty is the necessary coallocation and coscheduling of communicating components. Here we will consider a particular class of hybrid iterative methods. This will serve us as a case study and will be presented in the next section.

3 Case study

3.1 Hybrid iterative methods for linear algebra

Hybrid methods combine several different numerical methods or several copies of the same method parameterized differently to solve efficiently some numerical scientific problems. For example, both convergence acceleration techniques and preconditioning methods could be used to develop a hybrid method using the first way. An asynchronous parallel hybrid method has some properties such as asynchronous communications between its coarse grain subtasks, fault tolerance and dynamic load balancing which make this kind of methods well-adapted to

the Grid computational environments. The asynchronous hybrid algorithms can be easily implemented on a cluster of heterogeneous machines or on a Grid as it exhibits a coarse grain parallelism. These machines can be sequential, vector, or parallel. The number of iterations to convergence of the main process of a hybrid method can be reduced by combining results from other processes at runtime.

Each collaborative copy of a method, taking part in such hybrid computation is called a co-method and can be represented by a component. The natural parallelism of these components constituting a hybrid method can be different. An example of the second kind of the hybrid methods to compute a few eigenpairs of a large sparse non-hermitian matrix is the multiple explicitly restarted Arnoldi method (Multiple ERAM or MERAM) [3]. This method is based on a multiple projection of the explicitly restarted Arnoldi method (ERAM). Every collaborative component representing a co-method and taking part in such hybrid computation projects the initial problem in a different subspace. Each co-method calculates an approximated solution of the problem on its own subspace. The collaborative process is activated at the end of each iteration of every co-method. At this stage, the available intermediary results of the other co-methods are also considered in order to determine a better projection subspace for the next iteration. This process is depicted in Figure 1 in which $HR(1_{k_1}, 2_{k_2}, \ldots, \ell_{k_\ell}) = HR(U_{k_1}^{m_1}, \ldots, U_{k_\ell}^{m_\ell})$ denotes the hybrid restarting strategy taking into account $U_{k_i}^{m_i}$. Where $U_{k_i}^{m_i}$ is the set of the intermediary eigenvectors computed by the k_ith restart of the ith co-method (for $i = 1, \ldots, \ell$ and $k_i = 1, 2, \ldots$). In this figure, we suppose that we have to compute an approximation (λ^m, u^m) for the eigenpair (λ, u) of the matrix A. Thus, $U_{k_i}^{m_i}$ represents just the approximated eigenvector $u_{k_i}^{m_i}$ computed by the k_ith restart of the ith co-method.

Many algorithms based on the Krylov subspace methods, like ERAM, GMRES, Generalized Conjugate Residual method,... can be executed concurrently as a hybrid method. Indeed, once a co-method ends an iteration, the just computed information can be sent to the others to be incorporated in their next restarting strategy. Thus, each co-method can benefit from two types of results: its `own` results and the `remote` ones, issued from the other co-methods in collaborating computations.

3.2 Parallelism analysis and scheduling challenge

One of the great interests of the hybrid methods in linear algebra is their coarse grain parallelism. Nevertheless, the parallelization of these methods is a complex and challenging work due firstly to the existence of their two main levels of parallelism, and then to the heterogeneity of the architectures being used as their execution support. The first level parallelism is that one inter co-methods constituting a hybrid method. The second level is the parallelism intra co-method which can be exploited according to a data parallel, message passing or multithreading programming model. We concentrate here on the inter co-method parallelism.

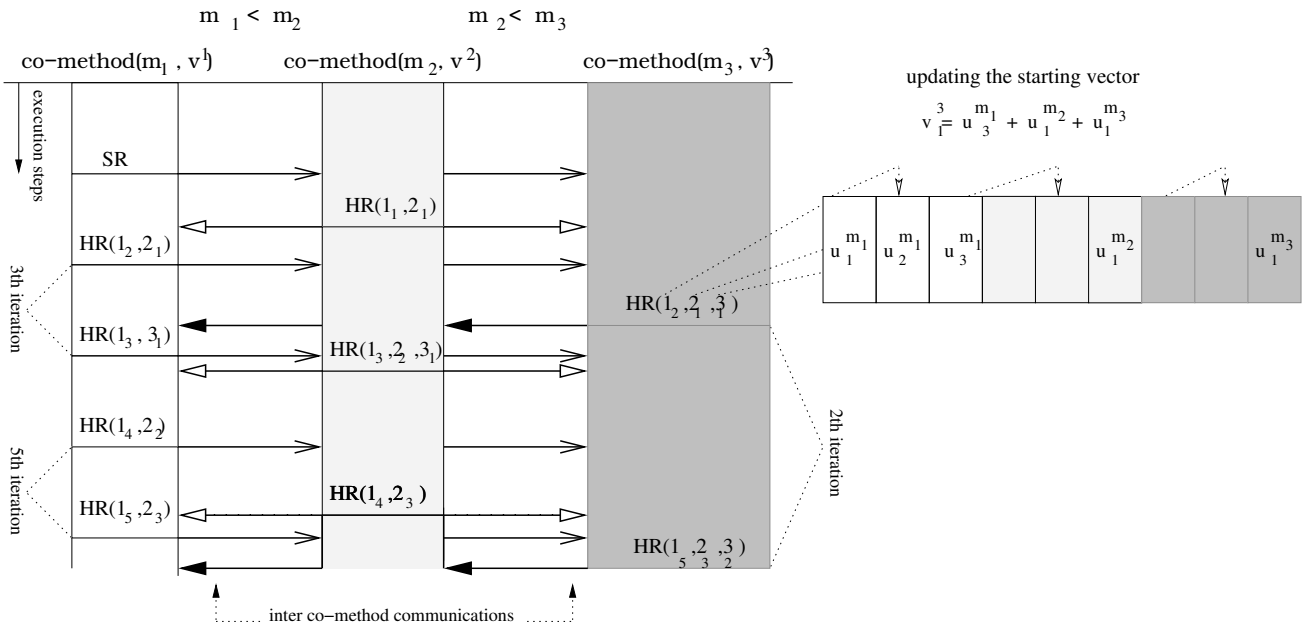


Fig. 1. A hybrid computation using $\ell = 3$ co-methods to compute an eigenpair (λ^m, u^m) of the matrix A. The co-method with the subspace size m_i and the initial guess v^i is denoted by co-method(m_i, v^i). HR and SR represent the hybrid and simple restarts.

Hybrid methods are well adapted for Grid computing. Different parallel components are just to be distributed to different resources. Nevertheless, the inter-component communications are asynchronous and difficult to be represented in a workflow model. Moreover, convergence detection introduces the necessity of interruption barriers between components. We will extend the workflow programming model to allow such asynchronous algorithms based, for example, on POP-C++ programming model [10]. Moreover, the components have to be simultaneously executed in order to collaborate. We exemplify this scheduling problem in the next subsection.

3.3 Scheduling hybrid methods: a basic example

Let us consider the same example described in Table 1. We suppose now to have three components which have to collaborate. Each component will use all available nodes on the allocated resource to maximize intra-parallelism work and therefore, to minimize iteration duration. We assume that the library L_2 is needed in order to run this collaborative work; all resources (R_1, R_2 and R_3) are thus admissible for running the components.

In a collaborative work, we have, as described in the cost model, simultaneity relations expressing the need to make all components running at the same time. In this example, those relations are $C_1 \simeq C_2$ and $C_2 \simeq C_3$.

A possible schedule for this hybrid method can be done as described in Table 4.

The metascheduler has relaxed the constraints of simultaneous starting time and proposes to wait until time 7 for taking benefit of the low cost period of

comp	resource	#proc	#start	#end	cost
C_1	R_1	12	6	15	3000
C_2	R_2	4	7	15	480
C_3	R_3	8	6	15	2000

Table 4. Example of schedules of co-methods in hybrid methods

R_2. C_1 and C_3 could begin a bit earlier without collaborating with C_2 at first. Maybe the use of only two co-methods during 1 time unit at first is not profitable. However, it is possible to set another schedule with all co-methods starting at time 7.

In the next section, we describe a programming environment well adapted to develop such hybrid methods on the Grid.

4 A candidate programming environment for developing hybrid methods on the Grid: POP-C++

4.1 Overview

The *POP-C++ programming environment* has been built to provide Grid programming facilities which greatly ease the development of parallel applications on the Grid. Figure 2 presents the layers of the POP-C++ architecture. The architecture supports the Grid-enabled application development at different levels, from the programming language for writing applications to the runtime system for executing applications on the Grid.

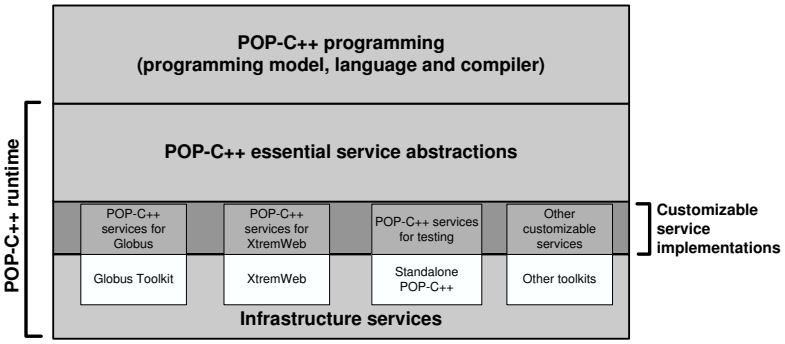


Fig. 2. The layered architecture of POP-C++ system

The POP-C++ runtime system consists of the infrastructure service layer managed by some Grid toolkits (e.g. Globus Toolkit or Unicore), the POP-C++ service layer to interface with the Grid infrastructures, and the POP-C++ essential service abstractions layer that provides a well defined abstract interface for the programming layer to access low-level services such as the resource discovery, the resource reservation or the object execution. The resource discovery and

reservation can be implemented using our proposed scheduling policy. Details of the POP-C++ runtime are described in [9].

POP-C++ programming, on top of the architecture, is the most important layer that provides necessary supports for developing Grid-enabled object-oriented applications based on the *parallel object model*.

4.2 POP-C++ programming model

The original *parallel object* model used in POP-C++ is the combination of powerful features of object-oriented programming and of high-level distributed programming capabilities. The model is based on the simple idea that objects are suitable structures to encapsulate and to distribute heterogeneous data and computing elements over the Grid. Programmers can guide the resource allocation for each object by describing their high-level resource requirements through the *object description*. The object creation process, supported by the POP-C++ runtime system, is transparent to programmers. Both inter-object and intra-object parallelism are supported through various original method invocation semantics. We intend to exploit POP-C++ objects and their descriptions to define components of hybrid methods and their timing constraints. Inter-object communications will be exploited for asynchronous inter co-methods communications.

The POP-C++ programming language extends C++ to support the parallel object model with just few new keywords for parallel object class declarations. Details of POP-C++ programming model are described in [8, 10]. With POP-C++, writing a Grid-enabled application becomes as simple as writing a sequential C++ application.

4.3 Parallel objects to capture components

One difficulty to develop and to deploy the component-based workflow model on the proposed scheduling system is the way to integrate resource requirements into each component. POP-C++ can help resolve this difficulty through its *object description* that allows programmers to describe their high level resource requirements such as the number of CPUs, the computing performance, the network bandwidth, etc. Although components can be implemented using any programming language, they are, in essence, very similar to POP-C++ objects. Nevertheless, the advantage of POP-C++ components is the ability to deduce all resource requirements of the components from their internal parallel structures. The proposed scheduling approach is well adapted to components written in POP-C++ but further study needs to be conducted in order to allow the POP-C++ compiler to automatically generate resources requirement of components.

5 Conclusion

In this paper, we have presented the problematic of scheduling intelligently some particular workflows on Grids. We have focus on a particular class of hybrid itera-

tive methods, which present collaborative asynchronous relations between coarse components. We have described a proposal for a generic cost model which can be a basis for the negotiation between a user and a metascheduler. We have investigated the feasibility of using the object-oriented programming environment POP-C++ for implementing and scheduling such hybrid methods on a computational Grid. Work is under way to implement and schedule an hybrid iterative method using the cost and evaluation models described in this paper and the programming environment POP-C++.

Acknowledgements

The authors wish to thank the European Commission for its support under the FP6 Network Of Excellence CoreGRID, which has permitted this joint work, and express their gratitude to the referees for their valuable comments.

References

1. J.M. Bahi, S. Contassot-Vivier and R. Couturier: *Asynchronism for iterative algorithms in a global computing environment.* HPCS'02, IEEE Computer Society Press, 2002.
2. M. Aldunicci, S. Campa, M. Coppola, M. Danuletto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, C. Zoccolo: *Components for high-performance grid programming in GRID.IT, in Components models and Systems for grid applications* V. Getov and T.Kielmann Eds, Springer, 2005, 19–38.
3. N. Emad, S. G. Petiton and G. Edjlali: *Multiple explicitly restarted Arnoldi method for solving large eigenproblems.* SIAM Journal on scientific computing SJSC, Volume 27, Number 1, pp. 253-277(2005)
4. R. Gruber, P.Volgers, A. De Vita, M. Stengel, T-M Tran: *Parameterisation to tailor commodity clusters to applications.* Future Generation Comp. Syst., 19,1,111-120,2003
5. H. He, G. Bergère and S. G. Petiton: *A Hybrid GMRES-LS-Arnoldi method to accelerate the parallel solution of linear systems* . Future Generation Comp. Syst., 19,1,111-120, 2003
6. V. Keller, K. Cristiano, R. Gruber, T-M Tran, P. Kuonen, P. Wieder, W. Ziegler, S. Maffioletti, N. Nellari, M-C Sawley: *Integration of ISS into the VIOLA Meta-scheduling Environment.* Computer and Mathematics with applications, 2005
7. M.S. Sadiq, Y. Habib: *Iterative computer algorithms in engineering: solving combinatorial optimization problems.* Wiley, 2000
8. T.A Nguyen, P. Kuonen: *ParoC++: A Requirement-driven Parallel Object-oriented Programming Language.* Proc. of the 8th International Workshop on High-Level Programming Models and Supportive Environments/IPDPS, 2003
9. T.A. Nguyen: An Object-oriented model for adaptive high performance computing on the computational Grid. PhD thesis, Swiss Federal Institute of Technology-Lausanne, 2004
10. T.A Nguyen, P. Kuonen: *Programming the Grid with POP-C++.* Future Generation Computer Systems, submitted 2005

Multi-criteria Grid Resource Management using Performance Prediction Techniques

Krzysztof Kurowski[1], Ariel Oleksiak[1], Jarek Nabrzyski[1],
Agnieszka Kwiecień[2], Marcin Wojtkiewicz[2], Maciej Dyczkowski[2],
Francesc Guim[3], Julita Corbalan[3], Jesus Labarta[3]

[1] Poznań Supercomputing and Networking Center
{krzysztof.kurowski,ariel,naber}@man.poznan.pl
[2] Wrocław Center for Networking and Supercomputing, Wrocław University of Technology
{agnieszka.kwiecien, marcin.wojtkiewicz, maciej.dyczkowski}@pwr.wroc.pl
[3] Computer Architecture Department, Universitat Politècnica de Catalunya
{fguim,juli,jesus}@ac.upc.edu

Abstract. To date, many of existing Grid resource brokers make their decisions concerning selection of the best resources for computational jobs using basic resource parameters such as, for instance, load. This approach may often be insufficient. Estimations of job start and execution times are needed in order to make more adequate decisions and to provide better quality of service for end-users. Nevertheless, due to heterogeneity of Grids and often incomplete information available the results of performance prediction methods may be very inaccurate. Therefore, estimations of prediction errors should be also taken into consideration during a resource selection phase. We present in this paper the multi-criteria resource selection method based on estimations of job start and execution times, and prediction errors. To this end, we use GRMS [28] and GPRES tools. Tests have been conducted based on workload traces which were recorded from a parallel machine at UPC. These traces cover 3 years of job information as recorded by the LoadLeveler batch management systems. We show that the presented method can considerably improve the efficiency of resource selection decisions.

1 Introduction

In computational Grids intelligent and efficient methods of resource management are essential to provide easy access to resources and to allow users to make the most of Grid capabilities. Resource assignment decisions should be made by Grid resource brokers automatically and based on user requirements. At the same time the underlying complexity and heterogeneity should be hidden. Of course, the goal of Grid resource management methods is also to provide a high overall performance. Depending on objectives of the Virtual Organization (VO) and preferences of end-users Grid resource brokers may attempt to maximize the overall job throughput, resource utilization, performance of applications etc.

Most of existing available resource management tools use general approaches such as load balancing ([25]), matchmaking (e.g. Condor [26]), computational economy

models (Nimrod [27]), or multi-criteria resource selection (GRMS [28]). In practice, the evaluation and selection of resources is based on their characteristics such as load, CPU speed, number of jobs in the queue etc. However, these parameters can influence the actual performance of applications differently. End users may not know a priori accurate dependencies between these parameters and completion times of their applications. Therefore, available estimations of job start and run times may significantly improve resource broker decisions and, consequently, the performance of executed jobs.

Nevertheless, due to incomplete and imprecise information available, results of performance prediction methods may be accompanied by considerable errors (to see examples of exact error values please refer to [3,4]). The more distributed, heterogeneous, and complex environment the bigger predictions errors may appear. Thus, they should be estimated and taken into consideration by a Grid resource broker for evaluation of available resources.

In this paper, we present a method for resource evaluation and selection based on a multi-criteria decision support method that uses estimations of job start and run times. This method takes into account estimated prediction errors to improve decisions of the resource broker and to limit their negative influence on the performance.

The predicted job start- and run-times are generated by the Grid Prediction System (GPRES) developed within the SGIgrid[30] and Clusterix[31] projects. The multi-criteria resource selection method implemented in the Grid Resource Management System (GRMS) [23,24,28] has been used for the evaluation of knowledge obtained from the prediction system. We used a workload trace from UPC.

Sections of the paper are organized as follows. In Section 2, a brief description of the related activities concerning performance prediction and its exploitation in Grid scheduling is given. In Section 3 the workload used is described. The prediction system and algorithm used for generation of predictions is included in Section 4. Section 5 presents the algorithm for the multicriteria resource evaluation and utilization of the knowledge from the prediction system. Experiments, which we performed, and preliminary results are described in Section 6. Section 7 contains final conclusions and future work.

2 Related work

Prediction techniques can be applied in a wide area of issues related to Grid computing: from the short-term prediction of the resource performance to the prediction of the queue wait time [5]. Most of these predictions are oriented to the resource selection and job scheduling.

Prediction techniques can be classified into statistical, AI, and analytical. Statistical approached are based on applications that have been previously executed. They can be time series analysis [6,7,8], categorization [4,1,2,22]. In particular correlation and regression have been used to find dependencies between job parameters. Analytical techniques construct models by hand [9] or using automatic code instrumentation [10]. AI techniques use historical data and try to learn and classify the information in order to predict the future performance of resources or applications. AI techniques

are, for instance, classification (decision trees [11], neural networks [12]), clustering (k-means algorithm [13]), etc.

Predicted times are used to predict resource information to guide scheduling decisions. This scheduling can be oriented to load balancing when executing in heterogeneous resources [14,15], applied to resource selection [5, 22], or used when multiple requests are provided [16]. For instance, in [17] authors use the 10-second ahead predicted CPU information provided by NWS [18,8]. Many local scheduling policies, such as Least Work First (LWF) or Backfilling, also consider user provided or predicted execution time to make scheduling decisions [19, 20,21].

3 Workload

The workload trace file was obtained in a IBM SP2 System placed at the UPC. It has two different configurations: the IBM RS-6000 SP with 8*16 Nighthawk Power3 @375Mhz with 64 Gb RAM, and the IBM P630 9*4 p630 Power4 @1Ghz with 18 Gb RAM. A total of 336Gflops and 1.8TB of Hard Disk are available. All nodes are connected through an SP Switch2 operating at 500MB/sec. The operating system that they are running is an AIX 5.1 with the queue system Load Leveler.

The workload was obtained from Load Leveler history files that contained around three years of job executions (178.183 jobs). Through the Load Leveler API, we convert the workload history files that were in a binary format, to a trace file whose format is similar to those proposed [21]. Fields in the workload are: job name, group, username, memory consumed by the job, user time, total time (user+system), tasks created by the job, unshared memory in the data segment of a process, unshared stack size, involuntary context switches, voluntary context switches, finishing state, queue, submission date, dispatch time, and completion date. More details on the workload can be found in [29].

Analyzing the trace file we can see that total time for parallel jobs is approximately an order of magnitude bigger than the total time for sequential jobs, what means that in median they are consuming around 10 times more of CPU time. For both kind of jobs the dispersion of all the variables is considerable big, however in parallel jobs is also around an order of magnitude bigger. Parallel jobs are using around 72 times more memory than the sequential applications, also the IQR is bigger[1]. In general these variables are characterized by significant variance what can make their prediction difficult.

Users submit jobs with various levels of parallelism. However, there is an important amount of jobs that are sequential (23%). The more relevant parallel jobs that are consuming more resources belong to three main number of processor usage intervals: 5-15 processors (31% of the total jobs), 65-128 processors (29% of the total jobs) and 17-32 processors (13% of the total jobs).

In median all the submitted LoadLeveler scripts used to be executed one time with the same number of tasks. This fact could imply that this last variable would be not

[1] The IRQ is defined as IQR=Q3-Q1, where: Q1 is a value such as a the exactly only 25% of the observations are less than it, and the Q3 is a value such as exactly on 25% of the observations are greater than it's bigger on these first one.

significant to be used for forecasting. However those jobs that where executed with 5-16 and 65-128 processors are executed in general more than 5 times with the same number of tasks, and represent the 25 % of the submitted jobs. This can suggest that this variable may be is relevant.

4 Prediction System

This section provides a description of the prediction system that has been used for estimating start and completion times of the jobs. Grid Prediction System (GPRES) is constructed as an advisory expert system for resource brokers managing distributed environment, including computational Grids.

4.1 Architecture

The architecture of GPRES is based on the architecture of expert systems. With this approach the process of knowledge acquisition can be separated from the prediction. The Figure 1 illustrates the system architecture and how its components interact with each other.

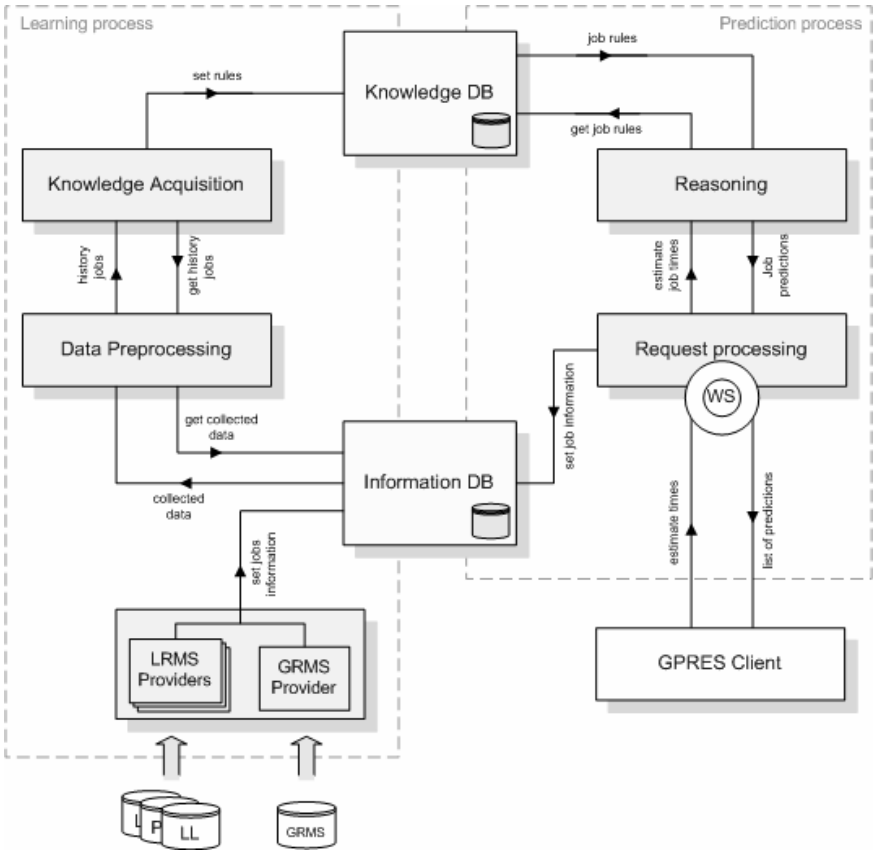


Fig. 1. Architecture of GPRES system

Data Providers are small components distributed in the Grid. They gather information about historical jobs from logs of GRMS and local resource management systems (LRMS, e.g. LSF, PBS, LL) and insert it into Information data base. After the

information is gathered the Data Preprocessing module prepares data for a knowledge acquisition. Jobs' parameters are unified and joined (if the information about one job comes from several different sources, e.g. LSF and GRMS). Such prepared data are used by the Knowledge Acquisition module to generate rules. The rules are inducted into the Knowledge Data Base. When an estimation request comes to GPRES the Request Processing module prepares all the incoming data (about a job and resources) for the reasoning. The Reasoning module selects rules from the Knowledge Data Base and generates the requested estimation.

4.2 Method

As in previous works [1, 2, 3, 4] we assumed that the information about historical jobs can be used to predict time characteristics of a new job. The main problem is to define the similarity of the jobs and to select appropriate parameters to evaluate it.

GPRES system uses a template-based approach. The template is a subset of job attributes, which are used to evaluate jobs' "similarity". The attributes for templates are generated from the historical information after tests.

The knowledge in the Knowledge Data Base is represented as rules:
IF $A_1 op v_1$ *AND* $A_2 op v_2$ *AND* ... *AND* $A_n op v_n$ *THEN* $d = d_i$, where $A_i \in A$, the set of condition attributes, v_i – values of condition attributes, $op \in \{=, \geq, \leq\}$, d_i – value of decision attribute, $i, n \in N$.

One rule is represented as one record in a data base. Several additional parameters are set for every rule: a minimum and maximum value of a decision attribute, standard deviation of a decision attribute, a mean error of previous predictions and a number of jobs used to generate the rule.

During the knowledge acquisition process the jobs are categorized according to templates. For every created category additional parameters are calculated. When the process is done the categories are inserted into the Knowledge Data Base as rules.

The prediction process uses the job and resource description as the input data. Job's categories are generated and the rules corresponding to categories are selected from the Knowledge Data Base. Then the best rule is selected and used to generate a prediction. Actually there are two methods of selecting the best rule available in GPRES. The first one prefers the most specific rule, with the best matching to condition attributes of the job. The second strategy prefers a rule generated from the highest number of history jobs. If both methods don't give the final selection, the rules are combined and the arithmetic mean of the decision attribute is returned.

5 Multi-criteria prediction-based resource selection

Knowledge acquired by the prediction techniques described above can be utilized in Grids, especially by resource brokers. Information concerning job run-times as well as a short-time future behavior of resources may be a significant factor in improving the scheduling decisions. A proposal of the multi-criteria scheduling broker that takes the advantage of history-based prediction information is presented in [22].

One of the simplest algorithms which requires the estimated job completion times is the Minimum Completion Time (MCT) algorithm. It assigns each job from a queue to resources that provide the earliest completion time for this job.

Algorithm MCT

```
For each job Ji from a queue
  For each resource Rj, at which this job can be
  executed
    Retrieve estimated completion time of job CJi,Rj
  Assign job Ji to resource Rbest so that
```

$$C_{J_i,R_{best}} = \min_{R_j}\left(C_{J_i,R_j}\right)$$

Nevertheless, apart from predicted times, the knowledge about potential prediction errors is needed. The knowledge coming from a prediction system shouldn't be limited only to the mean times of previously executed jobs that fit to a template. Therefore, we also consider minimum and maximum values, standard deviation, and estimated error (as explained in Section 4.2). These parameters should be taken into account during a selection of the most suitable resources. Of course, the mean time is the most important criterion, however, relative importance of all parameters depends on user preferences and/or characteristics of applications. For instance, certain applications (or user needs) may be very sensitive to delays, which can be caused by incorrectly estimated start and/or run times. In such case a standard deviation, minimum and maximum values become important. Therefore, a multi-criteria resource selection is needed to accurately handle these dependencies. General use of multi-criteria resource selection methods in Grids was described in [23].

In our case we used the functional model for aggregation of preferences. That means that we used a utility function and we ranked resources based on its values. In detail, criteria are aggregated for job J_i and resource R_j by the weighted sum given according to the following formula:

$$F_{J_i,R_j} = \frac{1}{\sum_{k=1}^{n} w_k} \sum_{k=1}^{n} w_k * c_k \qquad (1)$$

where the set of criteria C $(n=4)$ consists of the following metrics:
C_1 – mean completion time ($time_{Ji,Rj}$)
C_2 – standard deviation of completion time ($stdev_{Ji,Rj}$)
C_3 – difference between maximum and minimum values of completion time ($max_{Ji,Rj}$-$min_{Ji,Rj}$)
C_4 – estimated error of previous predictions ($err_{Ji,Rj}$)
and weights w_k that define the importance of the corresponding criteria.

This method can be considered as a modification of the MCT algorithm to a multi-criteria version. In this way possible errors and inaccuracy of estimations are taken into consideration in MCT. Instead of selection of a resource, at which a job completes earliest, the algorithm chooses resources characterized by the best values of the utility function $F_{Ji,Rj}$.

Multi-criteria MCT algorithm

For each job J_i from a queue
  For each resource R_j, at which this job can be executed
    Retrieve estimated completion time of job $C_{Ji,Rj}$ and $err_{Ji,Rj}$, $stdev_{Ji,Rj}$, $max_{Ji,Rj}$, $min_{Ji,Rj}$
    Calculate the utility function $F_{Ji,Rj}$
  Assign job J_i to resource R_{best} so that

$$F_{J_i,R_{best}} = \max_{R_j}\left(F_{J_i,R_j} \right)$$

6 Preliminary Results

There are two main hypothesis of this paper defined. First, use of knowledge about estimated job completion times may significantly improve resource selection decisions made by resource broker and, in this way, the performance of both particular applications and the whole VO. Nevertheless, estimated job completion times may be insufficient for effective resource management decisions. Therefore, the second hypothesis is that results of these decisions may be further improved by taking the advantage of information about possible uncertainty and inaccuracy of prediction.

In order to check these hypothesis we performed two major experiments. First, we compared results obtained by the MCT algorithm with a common approach based on the matchmaking technique (job was submitted to the first resource that met user's requirements). In the second experiment, we studied improvement of results of the prediction-based resource evaluation after application of knowledge about possible prediction errors. For both experiments the following metrics were compared: mean, worst, and best job completion time. The worst and best job completion values were calculated in the following way. First, for each application the worst/best job completion times have been found. Second, an average of these values was taken as the worst and best value for comparison.

5000 jobs from the workload were used to acquire knowledge by GPRES. Then 100 jobs from the workload were scheduled to appropriate queues using methods presented in Section 5.

The results of the comparison are presented in Figure 2. In general, it shows noticeable improvement of mean job completion times when the performance prediction method was used.

The least enhancement was obtained for the best job completion times. The multi-criteria MCT algorithm turned out to be the most useful for improvement of the worst completion times. Further study is needed to test the influence of relative importance of criteria on final results.

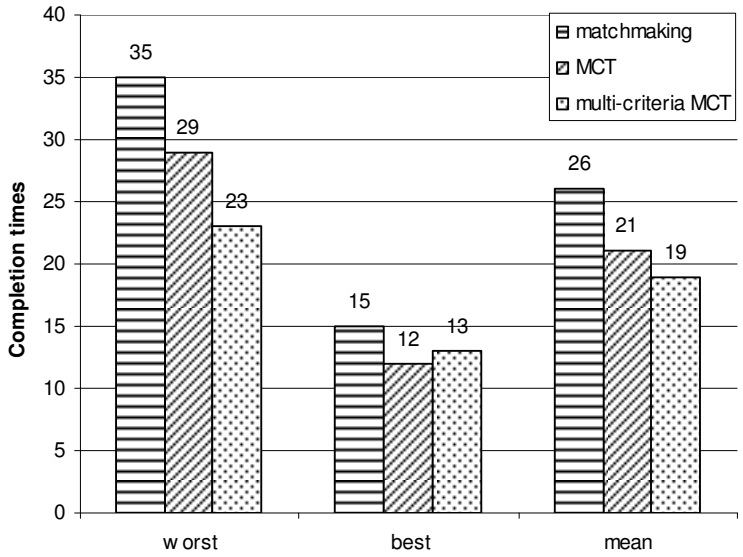


Fig. 2. Comparison of job completion times for matchmaking, MCT, and multi-criteria MCT algorithms

7 Conclusion

In this paper we proposed the multi-criteria resource evaluation method based on knowledge of job start- and run-times obtained from the prediction system. As a prediction system the GPRES tool was used. We exploited the method of multi-criteria evaluation of resources from GRMS.

The hypotheses assumed in the paper have been verified. Exploitation of the knowledge about performance prediction allowed a resource broker to make more efficient decisions. This was visible especially for mean values of job completion times.

Exploitation of knowledge about possible prediction errors brought another improvement of results. As we had supposed it improved mainly the worst job completion times. Thus, taking the advantage of knowledge about prediction errors we can limit number of job completion times that are significantly worst than estimated values. Moreover, we can tune the system by setting appropriate criteria weights depending on how reliable results we need and how sensitive to delays application are. For instance, certain users may accept "risky" resources (i.e. only the mean job completion time is important for them) while others may expect certain reliability (i.e. low ratio of strongly delayed jobs).

The advantage of performance prediction methods is less visible for strongly loaded resources because many jobs have to be executed at worse resources. This drawback could be partially eliminated by scheduling a set of jobs at the same time. This approach will be a subject of further research. Of course, information about possible prediction errors is the most useful in case of inaccurate predictions. If a resource broker uses high quality predictions, knowledge of estimated errors becomes less important.

Although a substantial improvement of the performance were shown, these results are rather still far from users' expectations. This is caused by, among others, a quality of available information. Most of workloads (including the LoadLeveler workload used for our study) do not contain such essential information as number of jobs in queues, size of input data, etc. Exploitation of more detailed and useful historical data is also foreseen as the future work on improving efficiency of Grid resource management based on performance prediction.

Acknowledgement

This work has been supported by the CoreGrid, network of excellence in "Foundations, Software Infrastructures and Applications for large scale distributed, Grid and Peer-to-Peer Technologies",, the Spanish Ministry of Science and Education under contract TIN2004-07739-C02-01, and SGIgrid and Clusterix projects funded by the Polish Ministry of Science.

References

1. Allen Downey, "Predicting Queue Times on Space-Sharing Parallel Computers". In International Parallel Processing Symposium, 1997.
2. Richard Gibbons. "A Historical Application Profiler for Use by Parallel Schedulers". Lecture Notes on Computer Science, pages 58-75, 1997.
3. Warren Smith, Valerie Taylor, Ian Foster. "Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance". In Proceedings of the IPPS/SPDP '99 Workshop on Job Scheduling Strategies for Parallel Processing.
4. Warren Smith, Valerie Taylor, Ian Foster. "Predicting Application Run-times Using Historical Information" In Proceedings IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, 1998.
5. I. Foster and C. Kesselman. Computational grids. In I. Foster and C. Kesselman, editors, The Grid: Blueprint for a New Computing Infrastructure, pages 15--52. Morgan Kaufmann, San Francisco, California, 1986.
6. R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared unix systems. In submitted to SIGMETRICS '99 (also available as UCSD Technical Report Number CS98-602), 1998.
7. P. Dinda. Online prediction of the running time of tasks. In Proc. 10th IEEE Symp. on High Performance Distributed Computing 2001
8. R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. Future Generation Computer Systems, 15 (5-6):757-768 1999
9. J. Schopf and F. Berman. Performance prediction in production environments. In Proceedings of IPPS/SPDP, 1998.
10. V. Taylor, X. Wu, J. Geisler, X. Li, z. Lan, M. Hereld, I. Judson, and R. Stevens. Prophesy: Automating the modeling process. In Proc. Of the Third International Workshop on Active Middleware Services, 2001.
11. J.R. Quinlan. Induction of decision trees. Machine Learning, pages 81-106, 1986
12. D.E.Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back propagating errors. Nature, 323:533-536, 1986

13. C.Darken, J.Moody: Fast adaptive K-Means Clustering: some Empirical Results, Proc. International Joint Conference on Neural Networks Vol II, San Diego, New York, IEEE Computer Scienc Press, pp.233-238, 1990.
14. H.J.Dail. A Modular Framework for Adaptive Scheduling in Grid Application Development Environments. Technical report CS2002-0698, Computer Science Department, University of California, San Diego, 2001
15. S. M. Figueira and F. Berman,Mapping Parallel Applications to Distributed Heterogeneous Systems, Department of Computer Science and Engineering, University of California, San Diego, TR - UCSD - CS96-484, 1996
16. K. Czajkowski, I. Foster, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., JSSPP Whorskshop. LNCS #1459 pages 62-68. 1997.
17. C. Liu, L. Yang, I. Foster, D. Angulo., Design and Evaluation of a Resource selection Framework for Grid Applications. In Proceedings if the Eleventh IEEE International Symposium on High-Performance Distributed Computing (HPDC 11), 2002
18. R. Wolski. Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In 6th High-Performance Distributed Computing, Aug. 1997.
19. D. Lifka, "The ANL/IBM SP scheduling system ". In Job Scheduling Strategies for Parallel Processing, D. G. Feitelson and L. Rudolph (eds.), pp. 295--303, Springer-Verlag, 1995. Lect. Notes Comput. Sci. vol. 949
20. D. G. Feitelson and A. Mu'alem Weil. Utilization and predictability in scheduling the IBM SP2 with backfilling. In Proc. 12th Int'l. Parallel Processing Symp., pages 542--546, Orlando, March 1998.
21. D.G.Feitelson. Parallel Workload Archive. http://www.cs.huji.ac.il/labs/parallel/workload
22. K. Kurowski, J. Nabrzyski, J. Pukacki, Predicting Job Execution Times in the Grid, in Proceedings of the 1st SGI 2000 International User Conference, Kraków, 2000
23. K. Kurowski, J. Nabrzyski, A. Oleksiak, and J, Węglarz,. ”Multicriteria Aspects of Grid Resource Management”, In *Grid Resource Management* edited by J. Nabrzyski, J. Schopf, and J. Węglarz, Kluwer Academic Publishers, Boston/Dordrecht/London, 2003.
24. Kurowski, K., Ludwiczak, B., Nabrzyski, J., Oleksiak, A., Pukacki, J.: “Improving Grid Level Throughput Using Job Migration and Rescheduling Techniques in GRMS”. *Scientific Programming. IOS Press*. Amsterdam The Netherlands 12:4 (2004) 263-273
25. B. A. Shirazi, A. R. Husson, and K. M. Kavi. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, 1995.
26. Condor project. http://www.cs.wisc.edu/condor.
27. D. Abramson, R. Buyya, and J. Giddy. A computational economy for Grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems*, 18(8), October 2002.
28. Grid Resource Management System (GRMS), http://www.gridlab.org/grms.
29. F.Guim, J. Corbalan, J. Labarta. Analyzing LoadLeveler historical information for performance prediction. In Proc. Of Jornadas de Paralelismo 2005. Granada, Spain
30. SGIgrid project. http://www.wcss.wroc.pl/pb/sgigrid/en/index.php
31. Clusterix project. http://www.clusterix.pcz.pl

Infrastructure for Adaptive Workflows in Semantic Grids

Laura Bocchi[1,2], Ondřej Krajíček[3,4], Martin Kuba[3,4]

[1] Istituto Nazionale di Fisica Nucleare - CNAF, Italy
[2] Dept. Computer Science, University of Bologna, Italy
[3] Institute of Computer Science, Masaryk University Brno, Czech Republic
[4] Faculty of Informatics, Masaryk University Brno, Czech Republic

Abstract. The paper describes the Grid-enabled infrastructure for service workflows based on concepts of Semantic Services and Semantic Grids. We consider workflow composition and tasks submission and describe implementation mechanisms and address potential drawbacks. Error handling in workflows is described using long-running transactions with foundations in process algebra.

1 Introduction

Today, applications of large-scale computing emerge in various fields of human knowledge. We focus on building the supporting infrastructure for applications in biomedicine, based on the concepts of Grid Computing and Service Oriented Architecture (SOA). Our proposed infrastructure, called SEAGRIN (*Semantic Adaptive Grid Infrastructure*) is build around the concept of Workflows. The biomedical background of SEAGRIN is described in [1].

The key motivation is to build a Grid infrastructure on top of existing applications, exposed as "classic" Web Services. For this purpose, SEAGRIN introduces a concept called *Overlay Grid*: a layer of Grid Services introduced over existing Web Services, which implement all the required functionality of workflow management (i.e. creation, task submission, error handling, etc.).

Subject of our collaboration is the definition and refinement of the SEAGRIN components for composition and scheduling of workflows. The refinement takes into account the existing results in automated service composition, mostly based on AI planning and deductive Theorem Proving. Our proposed infrastructure addresses error handling in workflow composition and execution. The requirements of error handling in SEAGRIN also present interesting analogies with those of e-business in the Web Service Architecture: in both contexts the use cases include the notion of long running activities in multi-domain, loosely coupled systems. We refer, in particular, to the notion of compensation in long running (or compensating) transactions, that are supported by most languages for business process definition in the Web service scenario (e.g., XLANG and BPEL). Compensations provide a weaker mechanism for error recovery, with respect to classic notion of rollback in ACID transactions but, on the other hand,

they do not force locking of resources. Our goal is to create a framework for error handling and recovery based on the above mentioned existing research.

In this paper we focus on two extensions. The former is related to the workflow execution. We consider the high level semantic for long running transactions described in [2] and its encoding in the asynchronous Pi Calculus. The provided formal model for long running transactions is easily usable to achieve a straightforward distributed implementation. The latter extension concerns automated service selection and composition. We consider, as an ontology for service representation, an extension of OWL-S that expresses the particular transactional support provided by a service. OWL-S [3] is an OWL-based ontology of services that has been developed as part of the DARPA Agent Markup Language Program (DAML). OWL (Web Ontology Language) [4] is a World Wide Web Consortium (W3C) recommendation for the definition of ontologies on the Web. The proposed approach refines the OWL-S extension we proposed in [5], by associating to a service description also the description of its compensation service. In this way, it is possible to select a service depending on the degree up to which it can be undone.

Section 1.1 presents an overview of the SEAGRIN architecture. Section 1.2 briefly outlines the notion of long running transaction and compensation. Section 2 describes current hints and issues in workflow management in SEAGRIN. Section 3 addresses the issue of error handling. Section 4 presents our conclusions.

1.1 SEAGRIN Overview

The key property of our infrastructure is that it is purely service oriented. We basically distinguish two kinds of services: *Primary Services* and *Infrastructure Services*.

The **Primary Services** encapsulate and expose the application logic. In biomedicine, these services provide interface to applications, information systems, communication portals and biomedical databases. These services may encapsulate resources, such as biomedical appliances and even human resources, such as experts communicating with the entire system via specialised portals. Workflow may consist of many of such primary services, implemented by various institutions, even geographically dislocated. SEAGRIN defines basic requirements these services must comply to. These are essentially having semantic annotation of service description and conforming to the WS-I Basic Profile. Depending on the nature of the encapsulated application/resource, we distinguish basic and long running services.

Basic Primary Services are ordinary Web Services, which follow the natural service communication paradigm. The service consumer issues a request to invoke a particular service operation on provided input data (and actively waits for the response or times out), the service processes the request and sends the result output data to the consumer. These services are **naturally synchronous**.

Long-running Primary Services offer a more complex communication paradigm. They encapsulate processing which is time-demanding or asynchronous in nature, such using a physical device, requesting information from human expert or

requesting service of a specialised laboratory. There is no way to generally specify the running time of operations of such services, it may vary greatly between individual invocations and it is often difficult to make any assumptions about it in general. As these services are still Web Services, the communication protocol is based on simple operation invocation. A simple communication protocol has been devised to support their asynchronous nature, we just provide a short overview of the protocol:

1. Service consumer issues the request and provides input data and *callback service binding*.
2. The service processes the request synchronously. The result of the operation is the status of the operation, the identity/location of the result data (in case of WSRF, this will be the identification of WS Resource), *time hint* suggesting when the results may be available (optional) and QoS characteristics of the result data (how long they will be available starting from the time they became available).
3. The consumer passively waits for the notification from the service, which should come in time specified by time hint. Service uses provided callback binding to issue the notification[1]. This notification contains the status of operation. If it is still not complete, it contains another time hint.
4. If the notification does not arrive in time, the consumer polls the service to detect possible service failure.

The **Infrastructure Services** provide building blocks for the infrastructure and functionality to create and control the workflow. These are fully fledged Grid Services and implement the overlay Grid.

The SEAGRIN overlay Grid[2] defines eight types of services, four of which are "creational" in nature, while the others may be considered "behavioural" (similarly to the distinction among Design Patterns). The creational services implement workflow building functionality and behavioural services implement the processing of workflow tasks. The actual types of services are:

Composer service provides tools for creating workflow *blueprints*. Workflow blueprint is a kind of a template which describes the workflow structure, the connections among services, lists all service alternatives (if any).

Builder service uses workflow blueprint to instantiate the workflow. Workflow instance has associated user identity and is used to do the actual processing.

Controller service provides workflow lifecycle management and interface for task submission. Since the workflow may have checkpointing or transactional capabilities, the controller may offer facilities for restarting the computation from checkpoint, etc.

Nest service provides factory-like capabilities and lifecycle management for all behavioural services (Wrappers, Dispatchers, Data Sources, Data Stores).

[1] There are existing standards in Web Service family, most notably WS-Notification, but these are still too complex and not well implemented.

[2] The design of the overlay Grid is still work in progress, it will probably be refined further in the future.

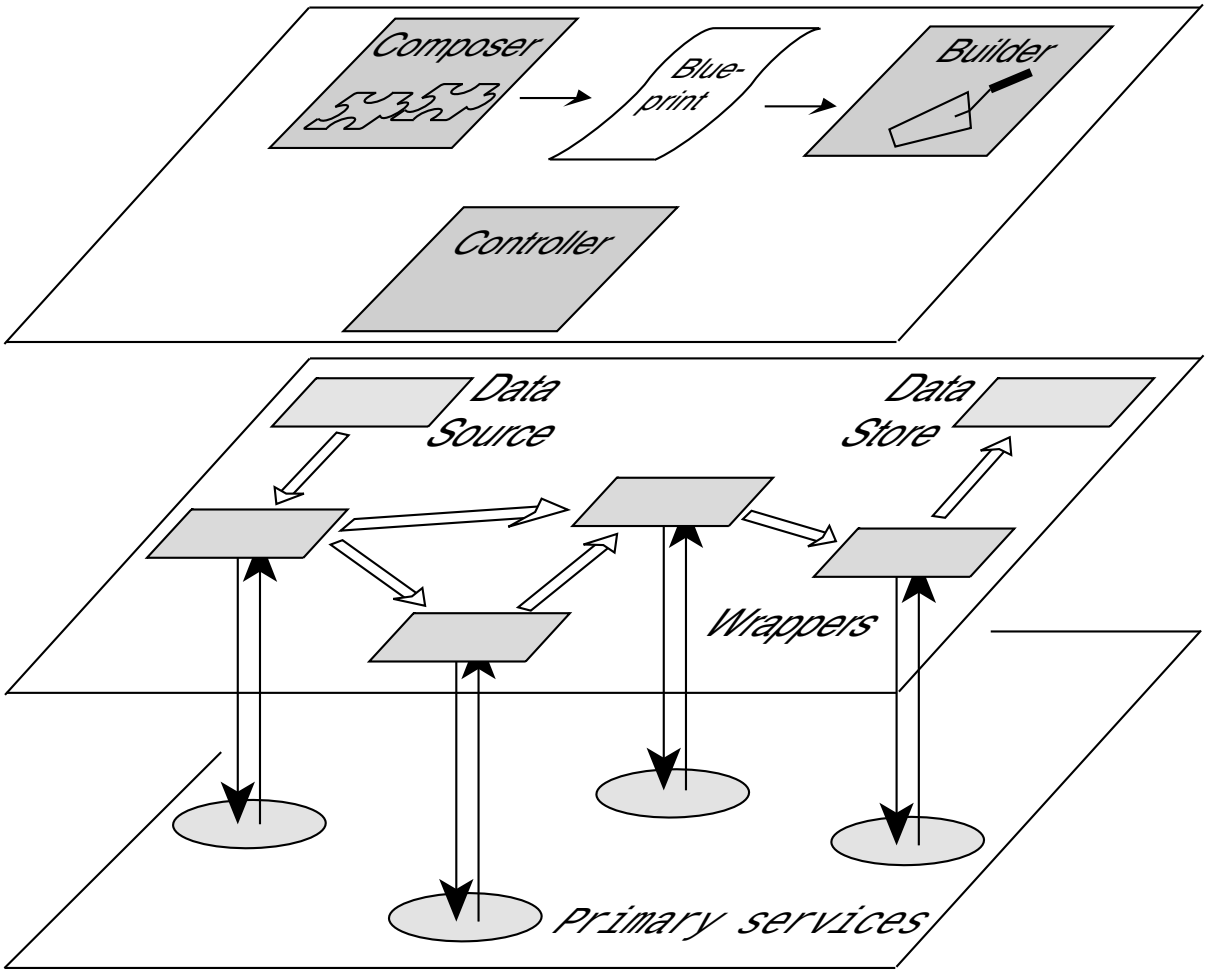


Fig. 1. SEAGRIN infrastructure - three layers formed by primary services, behavioural infrastructure services and creational infrastructure services

Wrapper service is used to encapsulate a primary service and communicate with other behavioural services. Execution of workflows is performed by means of direct communication among wrappers. Wrapper provides capabilities like translating messages from one XML schema to another (syntactic translations), converting between various data types (semantic translations). Wrapper may also implement monitoring of the primary service and in case of long running services, the wrapper encapsulates the asynchronous nature of such service.

Dispatcher implements "conditional dispatching", similarly to conditional statements in a high level programming language.

Data Store service is used to store data which are passed to it. Data Stores may encapsulate storage elements which are external to the system, such as storage of workflow "side effects" or subsequent results in external information systems, etc.

Data Source service is dual to the Data Store. It is used to retrieve data from external systems, such as information systems, relational databases or directory services. The data to retrieve may be specified by the messages passed

to the Data Source by the workflow. It means, the actual data retrieved may depend on the result of a task processing in a workflow.

1.2 Transactions in Loosely Coupled Environments

Within a SOA where services are *loosely coupled* and not always trusted, standard ACID transactions can turn out to be limiting factor. For example, Isolation usually means enforced locking of the resources used by each activity until the transaction commits (according to the two-phase locking protocol, which is usually employed). This is often not feasible in the management of long running activity involving resources of an external domain.

Long running transactions have been defined to deal with these issues. The principal difference is their weaker notion of rollback, referred to as *compensation*. A long running transaction neither blocks its resources nor performs temporary changes to the system: the actions executed are immediately visible and actual. If the transaction has to be *aborted* or undone after its completion, because of an error in the outer environment, the compensation is executed. It is important to notice that while the rollback is an integral part of the transaction execution, the compensation is an independent transaction executed afterwards and externally to its scope. The capability of the compensation to undo all the previously performed actions is relative to the particular context: in some cases it is impossible to undo all the effects (e.g., data deletion or e-mail sending).

2 Workflow Composition and Task Submission

One of the key problems of SEAGRIN is the task composition and workflow task submission. The workflow may be a simple sequence of services or it may use a more complex structure. It may be seen as a *network*, i.e. directed graph[3] with *source* (has *in-degree* 0) and *target* vertices (has *out-degree* 0) assuming that all nodes lie on some simple path from source to target. In the most general case the problem to solve is defined by users in terms of a *task definition*. Task definition describes the types of input data and/or desired results. It is the responsibility of the Composer service to create a blueprint for a workflow which transforms the input data to the desired output, and the responsibility of the Builder service to instantiate the workflow according to the blueprint. Actual data are submitted to the workflow by means of the Controller service in the form of a *task*. The task is then executed in an *execution flow*. The key difference between a task and execution flow is that while task defines the instances of input data, the execution flow represents the state of processing of a task, including all possible subsequent results, associated transaction, etc. Consequently, one task may be associated to more than one execution flow.

Based on the definition of the creational services, the workflow lifecycle can be broken into the following phases:

[3] The notion of dispatcher allows us to have cycles in a workflow, since the dispatcher may implement condition to end the cycle loop.

1. **Workflow Definition**: the end user submits a task definition to the Composer service. The Composer service returns a blueprint providing the possible solutions to the defined task.
2. **Workflow Refinement**: the end user refines the blueprint achieving a refined workflow blueprint. This blueprint may be stored in some kind of repository for later re-use.
3. **Workflow Creation**: the workflow is created using resulting workflow blueprint by the Builder service.
4. **Workflow Execution**: the user submits one or more actual tasks to the workflow, when the computation is complete, the user may use Controller to destroy the workflow.

The task definition submitted in *Workflow Definition* phase is essentially a query, used by the Composer service to perform the matchmaking, according to a precise service description. We consider service descriptions to be based on the OWL-S ontology. Services, according to OWL-S, are characterised by their functional properties (preconditions, postconditions), specification of input and output data and provided QoS features.

Based on the task definition, SEAGRIN provides a blueprint suggesting a number of solutions to the task. When a task definition is submitted, the Composer Service searches all known primary services and tries to match their properties to suggest possible workflow blueprints (i.e. solutions of the defined task) to the user.

In *Workflow Refinement* phase the workflow is refined by the end user, possibly helped by further invocations of the Composer service. For this purpose, the Composer service should provide workflow blueprint *validation*, i.e. it should be able to verify, that the workflow blueprint which has been manually altered by the end user is still instantiable.

In *Workflow Creation* phase blueprint serves as a template for creating workflow instances, which is a responsibility of the Builder service. When builder service builds the actual workflow (by communicating with Nests, which in turn instantiate the behavioural services – Wrappers, Dispatchers, Data Sources and Data Stores).

In *Workflow Execution* phase the user may use the Controller to submit tasks to the workflow and to monitor and control its status. The Controller manages the workflow execution, however the execution itself is decentralised and performed by autonomous communication among behavioural services. Example of workflow blueprint is shown in fig. 2.

3 Workflow Error Handling and Recovery

The possibility of executing workflow composed by different services makes error management a complex issue. Within a workflow, the management of a failure may naturally affect more than one service.

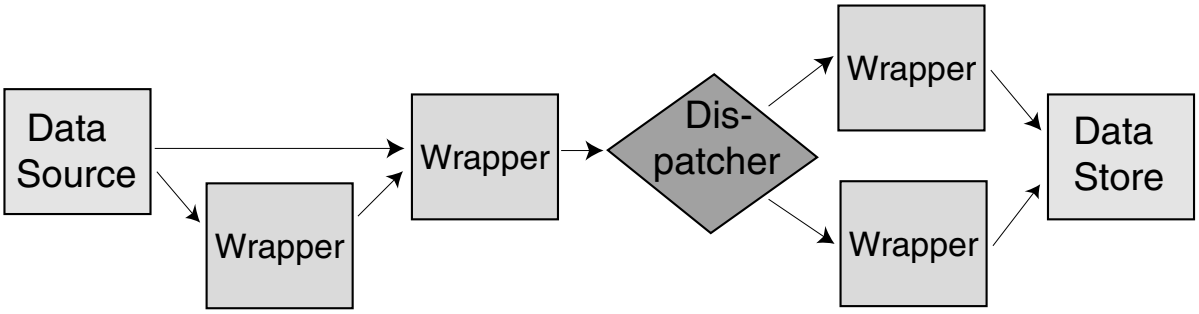


Fig. 2. Example of a Workflow

In Section 3.1 we propose an extension of the SEAGRIN with the notion of long running transaction presented in [2]. This extension introduces compensations as partial, ad hoc rollbacks for the invoked services. In Section 3.2, the notion of compensation, that is dependent from the service and the context, is associated to the service description. This enables a more fine grained service selection, considering also the semantic of the service compensation.

3.1 Long Running Transactions in SEAGRIN

The definition of infrastructure service is enhanced with a notion of long running transactions. While the long-running *services* are specialised class of primary services with unpredictable delay between request and response, the long running *transactions* are merely a concept how to implement error handling and the notion of transactions is considered to be recognised by all participating services.

Taking this into account, the long running transactions become workflow pattern for explicit (context-dependent and user-defined) error management. We base on the high level semantic defined in [2]. The advantages are: the usage of a formal approach and the possibility of using the provided encoding into the asynchronous pi calculus as a skeleton for a straightforward implementation.

Long-running transactions have two associated activities: the *failure* process and the *compensation* process. There are two kinds of transactions: those without inner transactions and the others. The first case is simpler: if the transaction fails, the failure process is executed. In the second case, if a transaction with inner transactions fails, the compensations of the inner transactions must be executed before activating the failure process of the enclosing transaction. Namely, after failure, the compensations can be activated in any possible order, independently of the order in which the corresponding transactions completed. Therefore, the programmer must explicitly describe inter-dependencies among compensations, to avoid undesired schedules by the run-time system.

Figure 3 illustrates the considered semantic for long running transactions. Transactions are informally represented in the figure as boxes. Each box is associated to two processes: the compensation process ranging over $C, C', \ldots$, the failure process ranging over $F, F', \ldots$. P, P' denote execution flows and *abort* represents the occurrence of a failure. The execution of the nested transaction of Figure 3 consists in the parallel execution of P and P', the concurrent execution

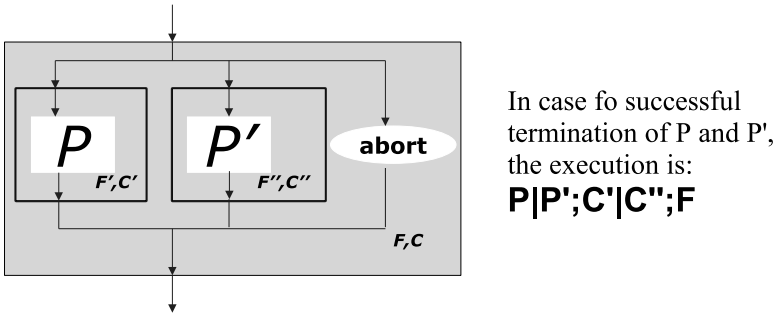


Fig. 3. An example of transaction behaviour in [2]

of C' and C'', and finally the execution of F. Notice that the abort is managed after the execution and the synchronisation of the concurrent execution flows $P \mid P'$. The operator ";" denotes sequential composition (i.e., $P_1; P_2$ requires P_1 to complete before starting the execution of P_2). Notice that sequence enables to model the synchronisation among parallel threads. For example the process $P_1 \mid P_2; P_3$ requires a synchronisation among P_1 and P_2 in order to start P_3.

The proposed solution is suitable to be implemented in the SEAGRIN architecture because the provided implementation with the asynchronous Pi calculus is distributed. In the encoding, different processes enclosed in a transaction are potentially distributed as the Wrappers managing the execution of the single primary services (e.g., processes/services P and Q in figure 4.

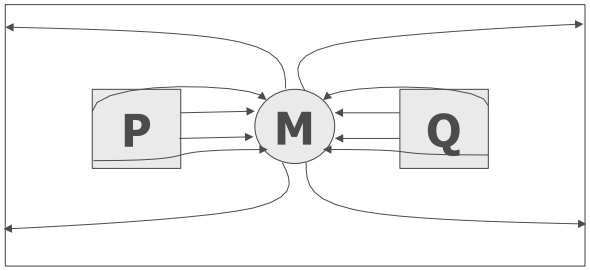


Fig. 4. The encoding of the synchronisation in [2]

Some example of how the presented mechanism can be used in SEAGRIN for explicit error handling are

- using, as a failure process, another execution of the same service,
- using, as a failure process, one of the alternatives that have been provided by the Composer Service during the phase of task composition,
- using, as a compensation process, a call to the compensation provided by the invoked service,
- defining an ad hoc compensation (e.g., de-allocate a previously allocated resource),

– using, as a compensation process, the empty process if it is not important (or not possible) to undo the effects of a completed service.

3.2 Compensation as a Parameter for Service Selection

In [5], the service description expressed by the OWL-S ontology is extended with the notion of supported *transactional behaviour*. The transactional behaviour gives a measure of "how much" a service can be undone after its completion. By extending the categorisation of [6], [5] introduces the following classification of the undo degree:

Unprotected services need not to be undone. An example is a service that is *referentially transparent*[4], i.e. with no permanent side effects and without any explicit resource allocation. When such service fails, there are no resources which could be stalled and no side effect to compensate for, in case of transaction abort.

Protected services could be compensated for. Conventional database operations are an example of protected actions, or resource reservation in Grid systems.

Real services cannot be undone. An example is an action performed on a real device such as an Instrument Element (IE) in a Grid. In biomedical Grid, this may be a blood probe, drug test evaluation, etc.

Semi-protected services offer partial (or *non-deterministic*) compensation capabilities. This means, that the the service operations could be compensated for, but the compensation could be only partial and may have dependencies, which are external and completely independent from the system.

Negotiated compensation degree services provide capabilities for runtime negotiation of their compensation capabilities. Such services may advertise more than one of the defined degrees in their description and manifest the actual compensational behaviour depending on the requirements imposed by the system.

Introducing Semi-protected transactions in the description of a service does not add, per se, much information enabling to establish up to witch degree the transaction is real and protected. Whereas adding the notion of compensation to the description of a service can add much information. As our proposed infrastructure is purely service oriented, the compensation itself is a service defined and described in terms of its pre- and postconditions, schema of input and output data and QoS characteristics.

4 Conclusions

Our collaboration started from an analysis of the SEAGRIN architecture. SEAGRIN has been designed to provide an adaptive infrastructure for workflows

[4] Referential transparency is a property known from functional programming paradigm. Essentially, it means that the result of an operation depends only and solely on the input data.

based on Semantic Grids. This paper presented a discussion about three key problems, which are the handling of long running services, the implementation of transactional behaviour in the workflow execution and service composition. In our approach, we propose to address these issues by reusing existing solutions in the Web service scenario. For example we include in the SEAGRIN management of workflows executions, enacted by the Dispatcher Service, similar mechanisms to those provided by existing engines of Web service orchestration languages. As to service composition, our approach aims to apply existing research on semantic-based service matchmaking. In this paper we included the transactional behaviour (i.e., the type of transactional support and the compensation provided by a service), as a relevant feature to consider in service selection, thus in service composition.

5 Acknowledgements

This research is partially supported by a research intent "Optical Network of National Research and Its New Applications" (Ministry of Education, Czech Republic – MSM6383917201) and research project *MediGrid – methods and tools for GRID application in biomedicine* (Czech Academy of Sciences, grant T202090537).

References

1. M. Kuba et al. Semantic Grid Infrastructure for Applications in Biomedicine. In Proc. DATAKON 2005, pages 335-344, Brno, 2005. ISBN 80-210-3813-6
2. L. Bocchi, C. Laneve, and G. Zavattaro. A Calculus for Long Running Transactions. In Proc. 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems, volume 2884 of Lecture Notes in Computer Science, pages 124–138. 2003.
3. The OWL Services Coalition. OWL-S 1.0 Release. `http://www.daml.org/services/owl-s/1.0/`
4. S. Bechhofer, F. Harmelen, J. Hendler, and I. Horrocks, D. McGuinnes, P. Patel-Schneider and L. A. Stein. OWL Web Ontology Language Reference, W3C, 2004. http://www.w3.org/TR/owl-ref/
5. L. Bocchi, P. Ciancarini, and D. Rossi. Transactional Aspects in Semantic Based Discovery of Services. In Proc. COORDINATION 2005, volume 3454 of Lecture Notes in Computer Science, pages 283–297. 2005.
6. J. Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In Proc. Proceedings of Very Large Data Bases, 7th International Conference, pages 144–154. 1981.
7. WS-I Basic Profile, v1.1 `http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html`

A Proposal for a Generic Grid Scheduling Architecture *

N. Tonellotto[1], R. Yahyapour[2], and Philipp Wieder[3]

[1] Information Engineering Department, University of Pisa, and ISTI-CNR
56100 Pisa, Italy
nicola.tonellotto@isti.cnr.it
[2] Robotics Research Institute, University of Dortmund,
44221 Dortmund, Germany
ramin.yahyapour@udo.edu
[3] Central Institute for Applied Mathematics, Research Centre Jülich,
52425 Jülich, Germany
ph.wieder@fz-juelich.de

Abstract. In the past years, many Grids have been implemented and became a commodity systems in production environments. While several Grid scheduling systems have already been implemented, they still provide only "ad hoc" and domain-specific solutions to the problem of scheduling resources in a Grid. However, no common and generic Grid scheduling system has emerged yet. In this work we identify generic features of three common Grid scheduling scenarios, and we introduce a single entity that we call scheduling instance that can be used as a building block for the scheduling solutions presented. We identify the behavior that a scheduling instance must exhibit in order to be composed with other instances to build Grid scheduling systems discussed, and their interactions with other Grid functionalities. This work can be used as a foundation for designing common Grid scheduling infrastructures.

1 Introduction

The allocation and scheduling of applications on a set of heterogeneous, dynamically changing resources is a complex problem. There are still no common Grid scheduling strategies and systems available which serve all needs. The available implementations of scheduling systems depend on the specific architecture of the target computing platform and the application scenarios. The complexity of the applications and the user requirements on the one side and the system heterogeneity on the other don't permit to efficiently perform manually any scheduling procedure.

The task of scheduling applications does not only include the search for a suitable set of resources to run applications with regard to some user-dependent Quality of Service (QoS) requirements; moreover the scheduling system may be

* This paper includes work carried out jointly within the CoreGRID Network of Excellence funded by the European Commission's IST programme under grant #004265.

in charge of the coordination of time slots allocated on several resources to run the application. In addition dynamic changes of the status of resources must be considered. It is the task of the scheduling system to take all those aspects into account to efficiently run an application. Moreover, the scheduling system must execute these activities while balancing several optimization functions: one provided by the user with her objectives (e.g. cost, response-time) as well as other objectives represented by the resource providers (e.g. throughput, profit).

These problems increase the complexity of the allocation and scheduling problem. Note that Grid scheduling significantly differs from the conventional job scheduling on parallel computing system. Several Grid schedulers have been implemented in order to reduce the complexity of the problem for particular application scenarios. However, no common and generic Grid scheduler yet exists, and probably there will never be one as the particular scenarios will require dedicated scheduling strategies to run efficiently. Nevertheless several common aspects can be found in these existing Grid schedulers which lead to assumption that a generic architecture may be conceivable which not only simplifies the implementation of different scheduling but also provide an infrastructure for the interaction between these different systems. Ongoing work [7] in the Global Grid Forum is describing those common aspects, and starting from this analysis we propose a generic architecture describing how a generic Grid scheduler should behave.

In Section 2 we analyze three common Grid scheduling scenarios, namely Enterprise Grids, High Performance Computing Grids and Global Grids. In Section 3 we identify the generic characteristics of the previous scenarios and their interactions with other Grid entities/services. In Section 4 we introduce a single entity that we call scheduling instance that can be used as a building block for the scheduling architectures presented and we identify the behavior that this scheduling instance must exhibit in order to be composed with other instances to build the Grid scheduling systems discussed.

2 Grid Scheduling Scenarios

In this Section three common Grid scheduling scenarios are briefly presented. This list is neither complete nor exhaustive. However, it represents common architectures that are currently implemented in application-specific Grid systems, either in research or commercial environments.

2.1 Scenario I: Enterprise Grids

Enterprise Grids represent a scenario of commercial interest in which the available IT resources within a company are better exploited and the administrative overhead is lowered by the employment of Grid technologies. The resources are typically not owned by different providers and are therefore not part of different administrative domains. In this scenario we have a centralized scheduling

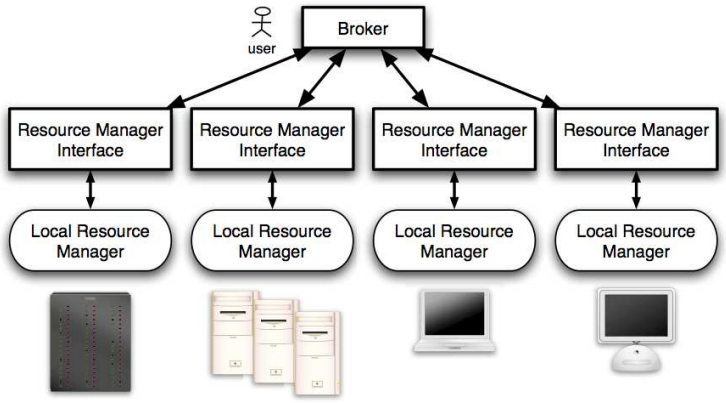


Fig. 1. Example of a scheduling infrastructure for Enterprise Grids

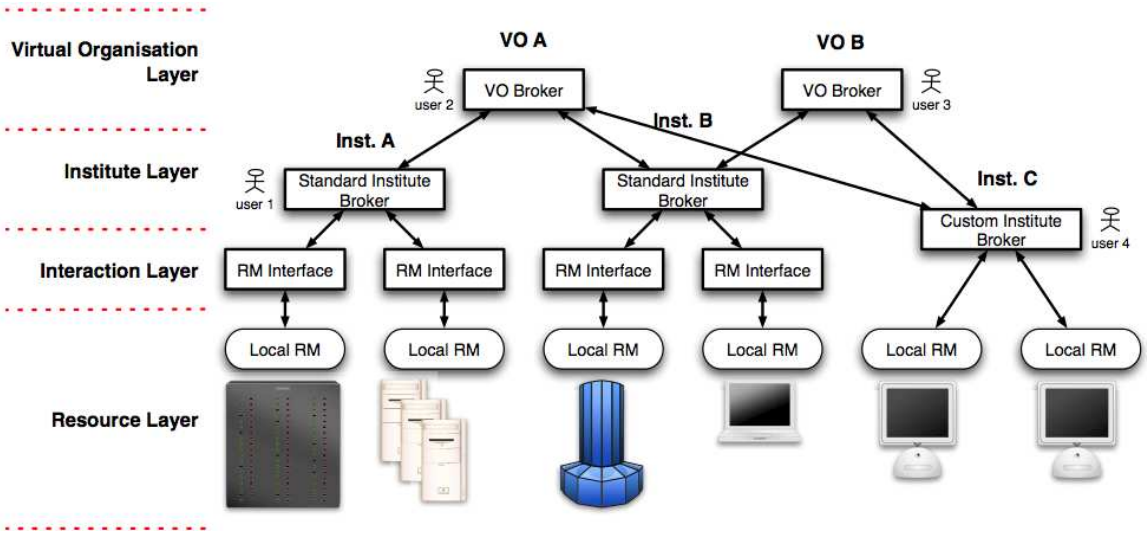


Fig. 2. Example of a scheduling infrastructure for HPC Grids

architecture, i.e. a central broker is the single access point to the whole infrastructure and manages directly the resource manager interfaces that interact with the local resource managers (see Figure 1). Every user must submit jobs to this centralized entity.

2.2 Scenario II: High Performance Computing Grids

High Performance Computing Grids represent a scenario in which different computing sites, e.g. scientific research labs, collaborate for joint research. Here, compute- and/or data-intensive applications are executed on the participating HPC computing resources that are usually large parallel computers or cluster systems. In this case the resources are part of several administrative domains, with their own policies and rules.

A user can submit jobs to the broker at institute or VO level. The brokers can split a scheduling problem into several sub-problems, or forward the whole problem to different brokers in the same VO.

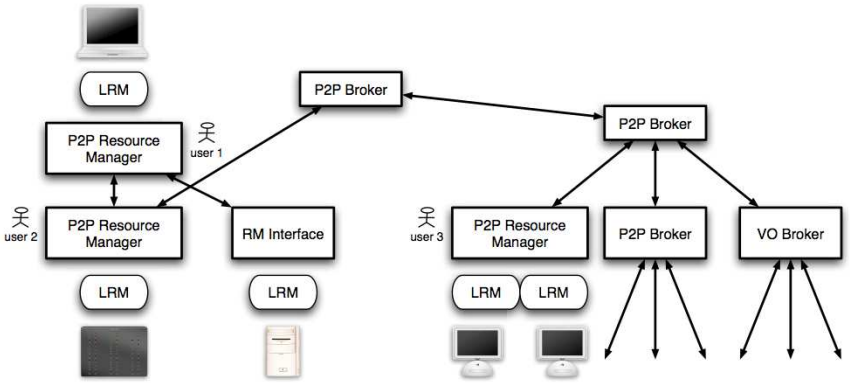


Fig. 3. Example of a scheduling infrastructure for Global Grids

2.3 Scenario III: Global Grids

Global Grids might comprise all kinds of resources, from single desktop machines to large-scale HPC machines, which are connected through a global Grid network. This scenario is the most general one, covering both cases illustrated above and introducing a fully decentralised architecture. Every Peer-to-Peer broker can accept jobs to be scheduled, as Figure 3 depicts.

3 Common Functions of Grid Scheduling

The three scenarios illustrated in the previous section show several entities interacting to perform scheduling. To solve scheduling problems, these entities can perform several tasks as described in [3, 4]. To perform them, they can interact with other entities/services, both external ones and those part of the GSA implementation. Exploiting the information presented in [7, 5], it is possible to identify a detailed list of core independent functions that can be used to build specific Grid scheduling systems. In the following a list of atomic, self-contained functions is presented; these functions can be part of any complex mechanism or process implemented in a generic Grid Scheduling Architecture (GSA).

- **Naming**: Every entity in play must have a unique identifier for interaction and routing of messages. Some mechanism must be in charge of assigning and tracking unique identifiers to the involved entities.
- **Security**: Every interaction between different un-trusted entities may need several security mechanisms. A scheduling entity may need to certify its identity when contacting another scheduling instance, when it is trying to collect sensible information about other entities (e.g. planned schedules of other instances), or to discover what interactions it is authorized to initiate. Moreover, the information flow may need secure transport and data integrity guarantees, and a user may need to be authorized to submit a problem to a scheduling system. The security functions are orthogonal to other ones, in the sense that every service needs security-related mechanisms.

- **Problem Submission**: The entity implementing this function is responsible to receive a job to be scheduled from a user and submit it to a scheduling component. At this level, the definition of job is intentionally vague, because it depends on the particular job submitted (e.g. a bag of tasks, a single executable, a workflow, a DAG). The job to be scheduled is provided using a user-defined language, and must be translated into a common description that is shared by some scheduling components.This description will therefore be exploited in the whole scheduling process. It should be able to identify scheduling related terms and to build agreement templates used by the scheduling instances to schedule the job.
- **Schedule Report**: An entity implementing this function must receive the the answer of the scheduling instance to a previously submitted problem and translate it into a representation consumable by the user.
- **Information**: A scheduling instance must have coherent access to static and dynamic information about resources characteristics (computational, data, networks, etc.), resource usage records, job characteristics, and, in general, services involved in the scheduling process. Moreover, it must be able to publish and update its own static and dynamic attributes to make them available to other scheduling instances. These attributes include allocation properties, local scheduling strategies, negotiation mechanism, local agreement templates and resource information relevant to the scheduling process [4]. It can be in addition useful to provide the capability to cache historical information.
- **Search**: This function can be exploited to perform optimized information gathering on resources. For example, in large scale Grids it can be neither important nor efficient to collect information about every resource, but just a subset of "good" candidate resources. Several search strategies can be implemented (e.g. "best fit" searches, P2P searches with caching, iterative searches). Every search should include at least two parameters: the number of records requested in the reply and a time-out for the search procedure.
- **Monitoring:** A scheduling infrastructure can monitor different attributes to perform its functions: it can be useful to monitor e.g. the status of an agreement or an allocation to check if they are respected, the execution of a job to undertake next scheduling or corrective actions, or the status of a scheduling description through the whole system for user feedback.
- **Forecasting**: In order to calculate a schedule it can be useful to rely on forecasting services to predict the values of the quantities needed to apply a scheduling strategy. These forecasts can be based on historical records, actual and/or planned values.
- **Performance Evaluation**: The description of a job to be scheduled can miss some information needed by the system to apply a scheduling strategy. In this case it can be possible to exploit performance evaluation methodologies based on the available job description in order to predict the unknown information.
- **Reservation**: In order to schedule complex jobs as workflows and co-allocated tasks, as well as jobs with guarantees, it is in general necessary to reserve

resources for particular time frames. The reservation of a resource can be obtained in several ways: automatically (because the local resource manager enforces it), on demand (only if explicitly requested from the user), etc. Moreover, the reservations can be restricted in time: for example only short-time reservations (i.e. with a finite time horizon) can be available. This function can require interaction with local resource managers and can be in charge of keeping information about allotted reservation and reserve new time frames on the resource(s).

- **Coallocation**: This function is in charge of the mechanisms needed to solve coallocation scheduling problems, in which strict constraints on the time frames of several reservations must be respected (e.g. the execution at the same time of two highly interacting tasks). It can rely on a low-level clock synchronization mechanism.
- **Planning**: When dealing with complex jobs (e.g. workflows) that need time-dependent access to and coordination of several objects like executables, data and network paths, a planning functionality, potentially built on top of a reservation service, is required.
- **Agreement**: In case quality of service guarantees concerning e.g. the allocation and execution time of a job must be considered, an agreement can be created and manipulated (e.g. accepted, rejected and modified) by the participating entities. A local resource manager can publish through its resource manager interface an agreement template regarding the jobs it can execute and a problem can include an agreement template regarding the guarantees that it is looking for.
- **Negotiation**: To reach an agreement the interacting partners may need to follow particular rules to exchange partial agreements to reach a final decision (e.g. who is in charge of providing the initial agreement template, who may modify what, etc.). This function should include a standard mechanism to implement several negotiation rules.
- **Execution**: This function is responsible to actually execute the scheduled jobs. It must interact with the local resource manager to perform the actions needed to run all the components of a job (e.g. staging, activation, execution, clean up). Usually it interacts with a monitoring system to control the status of the execution.
- **Banking**: The accounting/billing functionalities are performed by a banking system. It must provide interfaces to access accounting information, charging (in case of reservations or use of resources) and refunding (in case of agreement failures).
- **Translation**: The interaction with several services that can be implemented differently can force to translate information about the problem from the semantics of one system to the semantics of the other.
- **Data Management Access**: Data transfers can be included in the description of jobs. Although data management scheduling shows several similarities with job scheduling, it is considered a distinct, stand-alone functionality because the former shows significant differences compared to the latter (e.g. replica management and repository information) [2]. The implementation of

a scheduling system can need access to data management facilities to program data transfers with respect to planned job allocations, data availability and eligible costs. This functionality can rely on previously mentioned ones, like information management, search, agreement and negotiation.
- **Network Management Access**: Data transfers as well as job interactions can need particular network resources to respect guarantees on their execution. As in the previous case, due to its nature and complexity, network management is considered a stand-alone functionality that should be exploited by scheduling systems if needed [1, 6]. This functionality can rely on previously mentioned ones, like information management, search, agreement and negotiation.

4 Scheduling Instance

The different blocks in the previous examples can be considered particular implementations of a more general entity called scheduling instance. In this context, a scheduling instance is defined as a software entity that exhibits a standardized behavior with respect to the interactions with other software entities (which may be part of a GSA implementation or external services). The scheduling entities cooperate to provide, if possible, a solution to scheduling problems submitted by users, e.g. the selection, planning and reservation of resource allocations [4].

The scheduling instance is the basic building block of a scalable, modular architecture for scheduling jobs/applications in Grids. Its main function is to find a solution to a scheduling problem that it receives via a generic input interface. To do so, the scheduling instance needs to interact with local resource management systems that typically control the access to the resources. If a scheduling instance can find a solution for a submitted scheduling problem, the generated schedule is returned via a generic output interface.

From the previous examples it is possible to derive a high level model of operations for a generic set of cooperating scheduling instances. To provide a solution to a scheduling problem, a scheduling instance can exploit several options:

- It can try to solve the whole problem by itself with local resource managers that it is able to interact with.
- If it can partition the problem in several sub-problems, it can try to:
  1. solve some of the sub-problems, if possible,
  2. negotiate to forward the unsolved sub-problems to other,
  3. wait for potential solutions coming from other scheduling instances, or
  4. aggregate localized solutions to find a global solution for the original problem.
- If it cannot partition the problem or cannot find a solution by aggregating sub-problem solutions, it has two options:
  1. it can report back that it cannot find a solution or
  2. it can
     - negotiate to forward the whole problem to another, different scheduling instance or

- wait for a solution to be delivered by the instance the problem has been forwarded to.

A generic GSA will need to cover these behaviors, but actual implementations do not need to implement all of them. This model of operations is clearly modular, and permits to implement several scheduling infrastructures, like the ones depicted in the previous examples.

From them we can infer that a generic scheduling instance can exhibit the following abilities:

- interact with local resource managers;
- interact with external services that are not defined in the GSA;
- receive a scheduling problem (from other scheduling instances or external submission services), calculate a schedule, and return a scheduling decision (to the calling instance or an external service);
- split a problem in sub-problems, receive scheduling decisions and merge them into a new one;
- forward problems to other scheduling instances.

However, an instance might exhibit only a subset of such abilities. This depends on its interactions with other instances/services and its expected behavior (e.g. the ability to split and/or forward problems).

If a scheduling instance is able to cooperate with other instances, it must exhibit the ability to send problems or sub-problems, depending on the case, and receive scheduling results. Looking at such an instance, we call higher level instances the ones that are able to directly forward a problem to that instance, and lower level instances the ones that are able to directly accept a problem from that instance. A single instance must act as a decoupling entity between the actions performed at higher and lower levels: it is concerned neither with the previous instances through which the problem flows (i.e. it has been submitted by an external service or forwarded by other instances as a whole problem or as a sub-problem), nor with the actions that the following instances will undertake to solve the problem. Every instance will need to know just the problem it has to solve and the source of the original scheduling problem that helps to resolve, to avoid potential forwarding issues.

From a component point of view abilities as described above are expressed as interfaces. In general, the interfaces of a scheduling instance can be divided in two main categories: functional interfaces and non-functional interfaces. The former are necessary to enable the main behaviors of the scheduling instance, while the latter are concerned with the management of the instance itself (creation, destruction, status notification, etc.). We want to highlight that we considered only the functionalities that must be directly exploited to support a general scheduling architecture; for example, security services are from a functional point of view not strictly needed to schedule a job, so they are considered external services or non-functional interfaces. The functional interfaces that a scheduling instance can expose are depicted in Figure 4 and in detail described in the following:

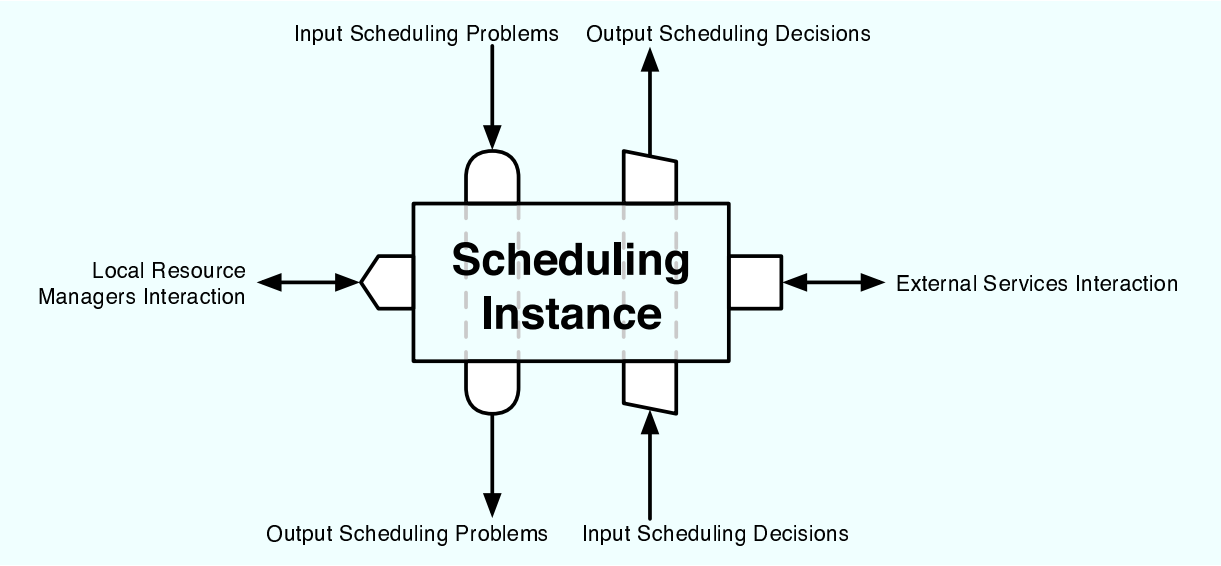


Fig. 4. Functional interfaces of a scheduling instance

Input Scheduling Problems Interface The methods of this interface are responsible to receive a description of a scheduling problem that must be solved, and start the scheduling process. This interface is not intended to accept jobs directly from users; rather an external submission service (e.g. portal or command line interface) can collect the scheduling problems described with a user-defined formalism, validate them and produce a neutral representation accepted as input by this interface. In this way, this interface is fully decoupled from external interactions and can be exploited to compose several scheduling instance as in the examples illustrated above, where an instance can forward a problem or submit a sub-problem to other instances using this interface. Every scheduling instance must implement this interface.

Output Scheduling Decisions Interface The methods of this interface are responsible to communicate the results of the scheduling process started earlier with a problem submission. Like the previous one, this interface is not intended to communicate the results directly to a user, rather to a visualization/reporting service. Again, we can exploit this decoupling in a modular way: if an instance received a submission from another one, it must use this interface to communicate the results to the submitting instance. Every scheduling instance must implement this interface.

Output Scheduling Problems Interface If an instance is able to forward a whole problem or partial sub-problems to other scheduling instances, it needs the methods of this interface to submit the problem to lower level instances.

Input Scheduling Decisions Interface If an instance is able to submit problems to other instances, it must wait until a scheduling decision is produced from the one which the problem was submitted to. The methods of this interface are responsible for the communication of the scheduling results from lower level instances.

Local Resource Managers Interface Sooner or later the scheduling process has to interact with local resource managers to allocate the jobs to the resources. While some scheduling instances can be dedicated to the "routing" of the problems, others interact directly with local resource managers to find suitable schedules, and propagate the answers in a neutral representation back to the entity submitting the scheduling problem. Different local resource managers can require different interaction interfaces.

External Services Interaction Interfaces If an instance must interact with an entity that is neither a local resource manager nor another scheduling instance, it needs an interface that permits to communicate with that external service that is exploited by the scheduling architecture. Different external services can require different interaction interfaces.

5 Conclusion

In this paper we discussed a general model for Grid scheduling. This model in based on a basic, modular component we called scheduling instance. Several scheduling instance implementations can be composed to build existing scheduling scenarios as well as new ones. The proposed model has no claim to be the most general one, but the authors consider this definition a good starting point to build a general Grid Scheduling Architecture that supports cooperation between different scheduling entities for arbitrary Grid resources. The future work will be directed towards further specifying the interaction of the Grid scheduling instance to other scheduling instances as well as to the other mentioned middleware services. The outcome of this work should yield a common Grid scheduling architecture that allows the integration of several different scheduling instances that can interact with eachother as well as be exchanged for domain-specific implementations.

References

1. V. Sander (Ed.). Networking Issues for Grid Infrastructure. GGF Document Series (GFD.37), 2004.
2. R. W. Moore. Operations for Access, Management, and Transport at Remote Sites. GGF Document Series (GFD.46), 2005.
3. J. M. Schopf. Ten Actions When Superscheduling. GGF Document Series (GFD.4), 2001.
4. U. Schwiegelshohn and R. Yahyapour. Attributes for Communication between Scheduling Instances. GGF Document Series (GFD.6), 2001.
5. U. Schwiegelshohn, R. Yahyapour, and Ph. Wieder. Resource management for future generation grids. Technical Report TR-0005, Institute on Scheduling and Resource Management, CoreGRID - Network of Excellence, May 2005.
6. D. Simeonidou and R. Nejabati (Eds.). Optical Network Infrastructure for Grid. GGF Document Series (GFD.36), 2004.
7. R. Yahyapour and Ph. Wieder (Eds.). Grid Scheduling Use Cases v1.2. GGF-GSA Working Draft, 2005.

Scheduling Workflows with Budget Constraints*

Eleni Tsiakkouri[1], Rizos Sakellariou[2], Henan Zhao[2], and Marios Dikaiakos[1]

[1] Department of Computer Science, University of Cyprus, Cyprus
[2] School of Computer Science, University of Manchester, U.K.

Abstract. Grids are emerging as a promising solution for resource and computation demanding applications. However, the heterogeneity of resources in Grid computing, complicates resource management and scheduling of applications. In addition, the commercialization of the Grid requires policies that can take into account user requirements, and budget considerations in particular. This paper considers a basic model for workflow applications modelled as Directed Acyclic Graphs (DAGs) and investigates heuristics that allow to schedule the nodes of the DAG (or tasks of a workflow) onto resources in a way that satisfies a budget constraint and is still optimized for overall time. Two different approaches are implemented, evaluated and presented using four different types of basic DAGs.

1 Introduction

In the context of Grid computing, a wide range of applications can be represented as workflows many of which can be modelled as Directed Acyclic Graphs (DAGs) [7, 9, 1, 5]. In this model, each node in the DAG represents an executable task (it could be an application component of the workflow). Each directed edge represents a precedence constraint between two tasks (data or control dependence). DAGs represent a model that helps build a schedule of the tasks onto resources in a way that precedence constraints are respected and the schedule is optimized. Virtually all existing work in the literature [6, 1, 8] aims to minimize the total execution time (length or makespan) of the schedule.

Although the minimization of an application's execution time might be an important user requirement, managing a Grid environment is a more complex task which may require policies that strike a balance between different (and often conflicting) requirements of users and resources. Existing Grid resource management systems are mainly driven by system-centric policies, which aim to optimize system-wide metrics of performance. However, it is envisaged that future fully deployed Grid environments will need to guarantee a certain level of service and employ user-centric policies driven by economic principles [2]. Of particular interest will be the resource access cost, since different resources, belonging to different organisations, may have different policies for charging.

* This work was supported by the CoreGRID European Network of Excellence, part of the European Commission's IST programme #004265

Clearly, users would like to pay a price which is commensurate to the budget they have available.

There has been little work examining issues related to budget constraints in a Grid context. The most relevant work is available in [3, 4], where it is demonstrated, through Grid simulation, how a scheduling algorithm can allocate jobs to machines at the same time satisfying constraints of Deadline and Budget. In this simulation, each job is considered to be a set of independent Gridlets (objects that contain all the information related to a job and its execution management details such as job length in million instructions, disk I/O operations, input and output file sizes and the job originator) [3]. Workflow types of applications, where jobs have precedence constraints, are not considered.

In this paper, we consider workflow applications that are modelled as DAGs. Instead of focussing only on makespan optimisation, as most existing studies have done [6, 8, 1], we also take into account a budget constraint. Each job, when running on a machine, costs some money. Thus, the overall aim is to find the shortest schedule for a given DAG and a given set of resources *without* exceeding the budget available. In this model, our emphasis is placed on the heuristics rather than the accurate modelling of a Grid environment; thus, we adopt a fairly static methodology in defining execution costs of the tasks of the DAG. However, as indicated by studies on workflow scheduling [1, 5, 9], it appears that heuristics performing best in a static environment (e.g., [6]) have the highest potential to perform best in a more accurately modelled Grid environment.

In order to solve the problem of scheduling optimally under a budget constraint, we propose two basic families of heuristics, which are evaluated in the paper. The idea in both approaches is to start from an assignment which has good performance under one of the two optimization criteria considered (that is, makespan and budget) and swap tasks between machines trying to optimize as much as possible for the other criterion. The first approach starts with an assignment of tasks onto machines that is optimized for makespan (using a standard algorithm for DAG scheduling onto heterogeneous resources, such as HEFT [8] or HBMCT [6]). As long as the budget is exceeded, the idea is to keep swapping tasks between machines by choosing first those tasks where the largest savings in terms of money will result in the smallest loss in terms of schedule length. We call this approach as LOSS. Conversely, a second approach starts with the cheapest assignment of tasks onto resources (that is, the one that requires the least money). As long as there is budget available, the idea is to keep swapping tasks between machines by choosing first those tasks where the largest benefits in terms of minimizing the makespan will be obtained for the smallest expense. We call this approach GAIN. Variations in how tasks are chosen result in different heuristics, which we evaluate in the paper.

The rest of the paper is organized as follows. Section 2 gives some background information about DAGs. In Section 3 we present the core algorithm proposed along with a description of the two approaches developed and some variants. In Section 4, we present experimental results that evaluate the two approaches. Finally, Section 5 concludes the paper.

2 Background

Following similar studies [1, 9, 7], the DAG model we adopt makes the following assumptions. Without loss of generality, we consider that a DAG starts with a single entry node and has a single exit node. Each node connects to other nodes with edges, which represent the node dependencies. Edges are annotated with a value, which indicates the amount of data that need to be communicated from a parent node to a child node. For each node the execution cost on each different machine available is given. In addition, the communication cost between machines is given. Traditional studies aim to assign tasks onto machines in such a way that the overall schedule length is minimized and precedence constraints are met. An example of a DAG and the schedule length produced using a well-known heuristic, HEFT [8] is shown in Figure 1. A number of other heuristics could be used too (see [6], for example). It is noted that in the example in the figure no task is ever assigned to machine M2. This is due to the high communication cost and since HEFT assigns tasks onto the machine that provides the earliest finish time, no task ever satisfies this condition.

The contribution of this paper relates to the extension of the traditional DAG model by adding (financial) cost to the available processors. The cost is measured in units of money. As a result, an additional constraint needs to be satisfied, namely that the overall financial cost of the schedule does not exceed a certain budget. We define the total cost (see Equation 2) as the sum of the costs of executing each task in the DAG onto a machine. This cost is calculated as the product of the execution time required by the task on the machine that has been assigned to, times the cost of this machine. This is shown in Equation 1, where $C_{i,j}$ is the cost of executing task i to machine j, $MachineCost_j$, is the cost (in money units) per unit of time to run something on machine j and $ExecutionTime_{i,j}$ is the time task i takes to execute on machine j.

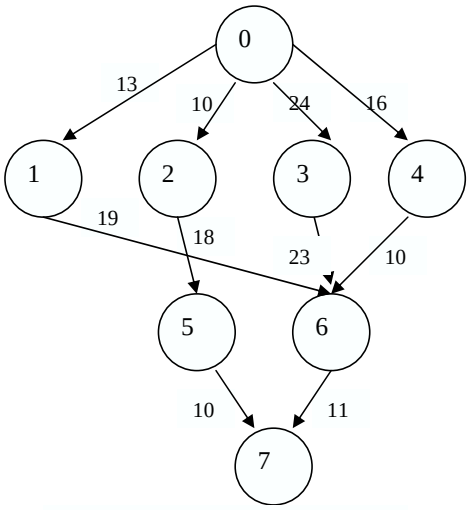
$$C_{i,j} = MachineCost_j \times ExecutionTime_{i,j} \tag{1}$$

$$TotalCost = \sum C_{i,j} \tag{2}$$

3 The Algorithm

3.1 Outline

The key idea of the algorithm proposed is to satisfy the budget constraint by finding the "best" affordable assignment possible. We define the "best assignment" as the assignment whose execution time is the minimum possible. The algorithm starts from a given assignment (schedule) and computes for each reassignment of each task to a different machine, a weight value associated with that particular change. A weight table is created for each task in the DAG and each machine. Two alternative approaches for computing the weight values are proposed depending on the two choices used for the starting assignment: either optimal for makespan (approach called LOSS), or cheapest (approach called GAIN); the two



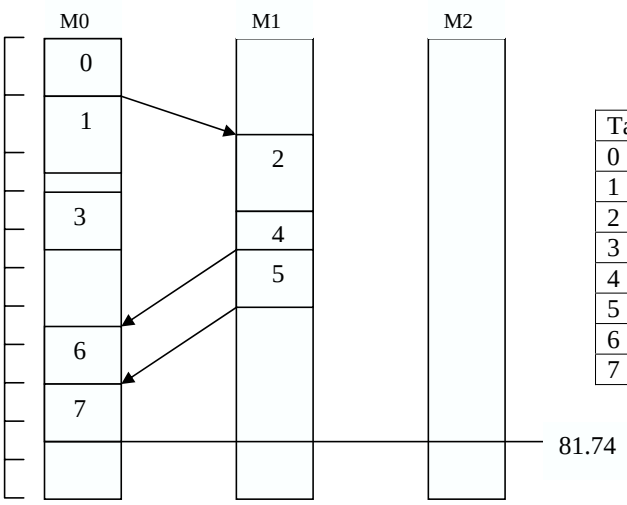
a) A DAG example

Task	M0	M1	M2
0	17.0	28.0	17.0
1	26.0	11.0	14.0
2	30.0	13.0	27.0
3	6.0	25.0	3.0
4	12.0	2.0	12.0
5	7.0	8.0	23.0
6	23.0	16.0	29.0
7	12.0	14.0	11.0

b) The computation cost of tasks on machines

Machines	Time for a data unit
M0-M1	1.607
M1-M2	0.9
M0-M2	3

c) Communication costs between the machines



d) Schedule derived by the HEFT algorithm

Task	Start Time	Finish Time
0	0.0	17.0
1	17.0	43.0
2	33.07	46.07
3	43.0	49.0
4	46.07	48.07
5	48.07	56.07
6	58.74	69.74
7	69.74	81.74

e) Start Time and Finish Time of each task in Schedule

Fig. 1. An Example of HEFT scheduling in a DAG workflow

approaches are described in more detail below. Using the weight table, tasks are repeatedly considered for possible reassignment to a machine, until the budget is exceeded. We illustrate the key steps of the algorithm in Table 1.

3.2 The LOSS Approach

The LOSS approach uses as a starting assignment the output assignment of either HEFT [8] or HBMCT [6] DAG scheduling algorithms. If the available budget is bigger or equal to the money cost required for this assignment then this assignment can be used straightaway. In all the other cases that the budget is less than the cost required for the starting assignment, the LOSS approach is invoked. The main aim of this approach is to make a change in the schedule (assignment)

```
Input: An application graph G and a schedule S produced by a scheduling algorithm H,
      Budget B
Cost-Time Scheduling Algorithm:

1) Get schedule S [] after running algorithm H.
2) Build an array A[task_number x machine_number]
3) Place a  ZEROs at cells A[T,M] where task T is assigned to machine M as indicated in S []
4) For each Task in  G
       for each Machine in G
           Compute the Weight value placed in A[Task,Machine]
        endfor
     endfor

5) While Cost of assignment < B
Find the a smallest or bigger (depending on approach used) value from A.
Value = A [i,j]
            Re-assign Task i to machine j in S[] and calculate new cost of assignment.
      endwhile
6) Return S
```

Table 1. The Basic Steps of the Proposed Scheduling Algorithm

given from, say HEFT, so that it will result in the minimum loss in execution time for the largest money savings. This means that the new schedule has an execution time close to the time the HEFT assignment would require but with less cost. In order to come up with such a re-assignment, the LOSS weight values for each task to each machine are computed as follows:

$$LossWeight(i,m) = \frac{T_{new} - T_{old}}{C_{old} - C_{new}} \tag{3}$$

where T_{old} is the time to execute task i on the machine assigned by HEFT, T_{new} is the time to execute Task i on machine m. C_{old}, respectively, is the cost of executing task i on the machine given by the HEFT assignment and C_{new} is the cost of executing task i on machine m. If C_{old} is less than or equal to C_{new} the value of $LossWeight$ is considered zero. The algorithm keeps trying re-assignments by considering the smallest values of the $LossWeight$ for all tasks and machines (step 5) of the algorithm in Table 1.

3.3 The GAIN Approach

The GAIN approach uses as a starting assignment the assignment that requires the less money. Each task is initially assigned to the machine that executes the task with the smallest cost. This assignment is called the Cheapest Assignment. In this variation of the algorithm, the idea is to change the Cheapest Assignment by keeping re-assigning tasks to the machine where there is going to be the biggest benefit in makespan for the smallest money cost. This is repeated until there is no more money available (budget exceeded). In a way similar to

Equation 3, weight values are computed as follows. It is noted that tasks are considered for reassignment starting with those that have the largest $GainWeight$ value.

$$GainWeight(i,m) = \frac{T_{old} - T_{new}}{C_{new} - C_{old}} \quad (4)$$

where T_{old}, T_{new}, C_{new}, C_{old} have exactly the same meaning as in the LOSS approach. Furthermore, if T_{new} is greater than T_{old} or C_{new} is equal to C_{old} we assign a weight value of zero.

3.4 Variants

For each of the two approaches above, we consider three different variants which relate to the way that the weights in Equations 3 and 4 are computed; these modifications result in slightly different versions of the heuristics. The three variants are:

- LOSS1 and GAIN1: in this case, the weights are computed exactly as described above.
- LOSS2 and GAIN2: in this case, the values of T_{old}, T_{new}, and C_{new}, C_{old} in Eqs 3 and 4 refer to the overall makespan and the overall cost, respectively, of the schedule and not the values associated with the individual tasks.
- LOSS3 and GAIN3: in this case, the weights in Equations 3 and 4 are recomputed each time a reassignment is made by the algorithm.

4 Experimental Results

4.1 Experiment Setup

The algorithm described in the previous section was incorporated in a tool developed at the University of Manchester, for the evaluation of different DAG scheduling algorithms [6, 7]. In order to evaluate each version of both approaches we run the heuristic proposed in this paper with four different types of DAGs used in the relevant literature [6, 7]: FFT, Fork-Join (denoted by FRJ), Laplace (denoted by LPL) and Random DAGs, generated as indicated in [10, 6], with around 100 nodes each. We run the heuristic proposed in the paper 100 times for each type of DAG and both approaches and their variants, and we considered the average values. In each case, we considered nine values for the possible budget, B, as follows:

$$B = C_{cheapest} + k \times (C_{HEFT} - C_{cheapest}), \quad (5)$$

where C_{HEFT} is the total cost of the assignment produced by HEFT and $C_{cheapest}$ is the cost of the cheapest assignment. The value of k varies between 0.1 and 0.9. Essentially, this approach allows us to consider values of budget that lie in ten equally distanced points between the money cost for the cheapest assignment and the money cost for HEFT. Clearly, values for budget outside those two ends are trivial to handle since they indicate that either there is no solution satisfying the given budget, or HEFT can provide a solution within the budget.

4.2 Metrics

Average Percentage Difference metric: In order to compare the quality of the schedule produced by the heuristic for each of the six variants and each type of DAG, and since 100 experiments are considered in each case, we normalize the schedule length (makespan) using the following formula:

$$\frac{T_{value} - T_{cheapest}}{T_{HEFT} - T_{cheapest}}, \tag{6}$$

where T_{value} is the makespan returned by the heuristic, $T_{cheapest}$ is the makespan of the cheapest assignment and T_{HEFT} is the makespan of HEFT. As a general rule, the makespan of $T_{cheapest}$ is expected to be the worst, hence, for comparison purposes, larger values in (6) indicate a shorter makespan. Since for each case we take 100 runs, the average value of the quantity above produces the *Average Percentage Difference* (APD) from the cheapest, that is,

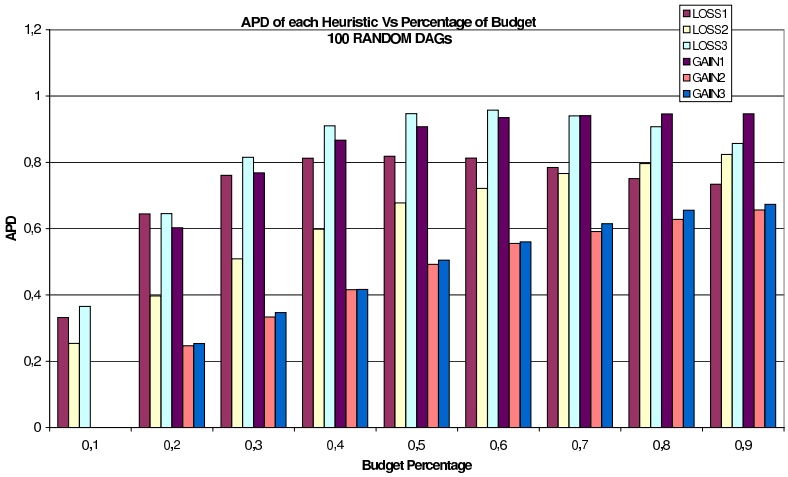
$$APD = \frac{1}{100} \sum_{i=1}^{100} \left(\frac{T^i_{value} - T^i_{cheapest}}{T^i_{HEFT} - T^i_{cheapest}} \right), \tag{7}$$

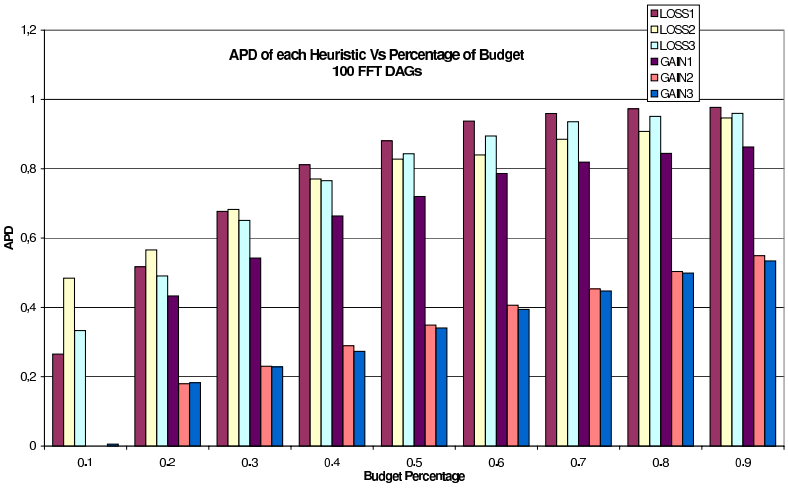
where the superscript i denotes the i-th run.

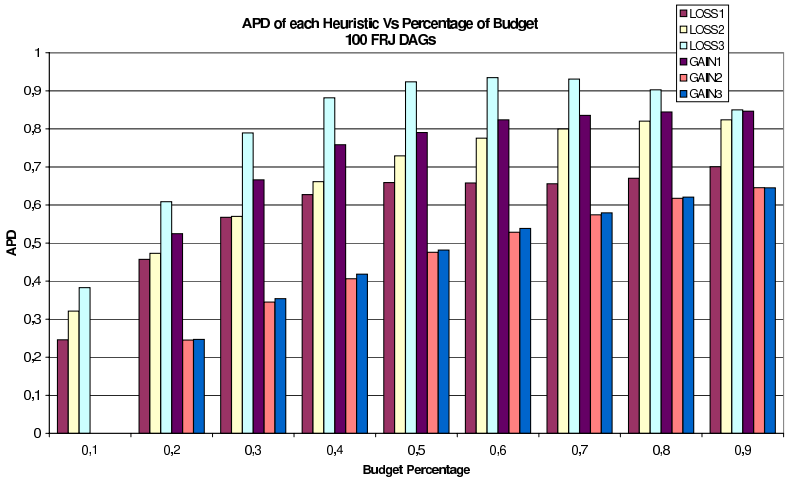
Results showing the APD for each different type of DAG, variant, and budget available (shown in terms of the value of k — see Equation 5) are presented in Figure 2. The graphs show the difference of the two approaches. The LOSS variants are generally better than the GAIN variants. This might be due to the fact that the starting basis of the LOSS approach is HEFT which produces a short makespan. Instead, the GAIN approach starts from the Cheapest Assignment whose makespan is typically long. However, from the experimental results we notice that in cases where the budget is close to the cheapest budget, the APD of the GAIN approaches becomes bigger than the APD of the LOSS approaches. This can be seen in Figure 2 for Random and FRJ DAGs.

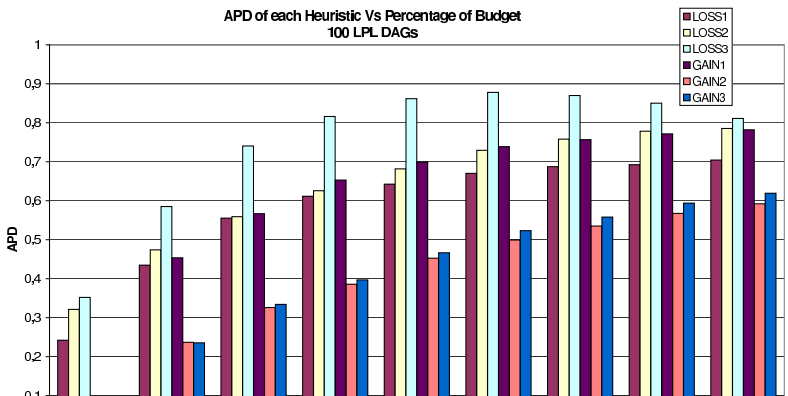
Execution Time for each Heuristic: To evaluate the performance of each version in both approaches we extracted for the experiments we carried out before, the execution time of the algorithm. The results are shown in Figure 3. Same as before, the execution time is the average value from 100 runs.

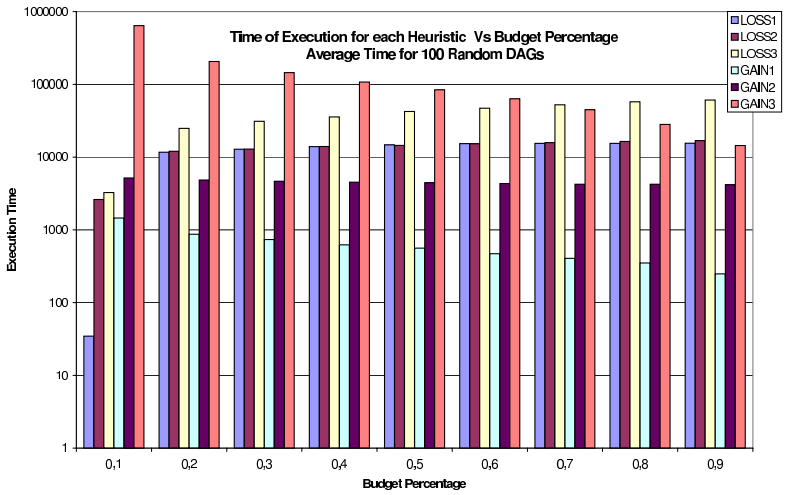
Observations: The above experiments indicate that the algorithm proposed in this paper is able to find affordable assignments with better makespan when the LOSS approach is applied, instead with the GAIN approach. The LOSS approach applies re-assignment to an assignment that is given by a good DAG scheduling heuristic, whereas in the GAIN approach the cheapest assignment is used which may have the worst makespan. However, in cases where the available budget is close to the cheapest budget, GAIN1 gives better makespan than LOSS1 or LOSS2. This can contribute to the optimisation in the performance of the heuristic.

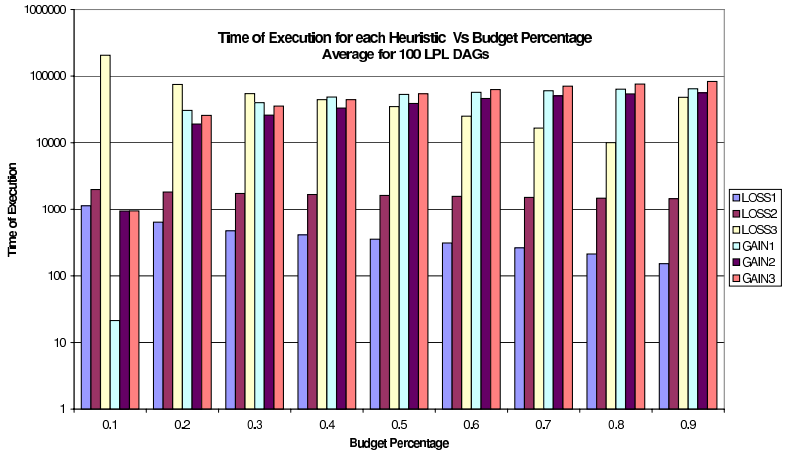

APD of each Heuristic Vs Percentage of Budget
100 RANDOM DAGs
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3
APD
Budget Percentage

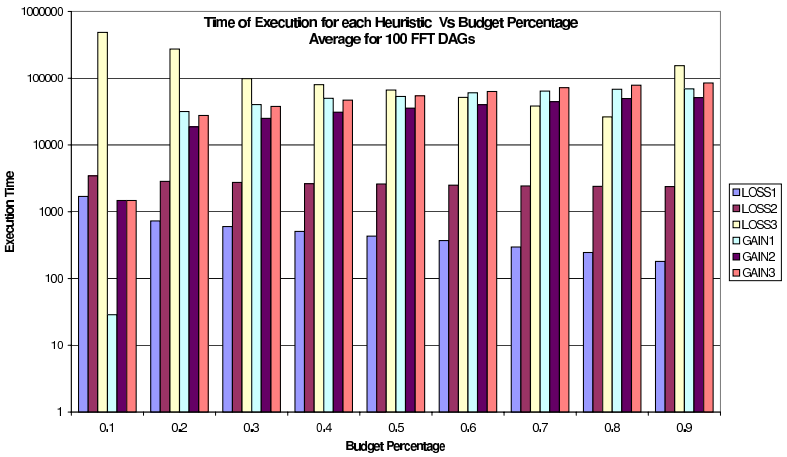

APD of each Heuristic Vs Percentage of Budget
100 FFT DAGs
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3
APD
Budget Percentage

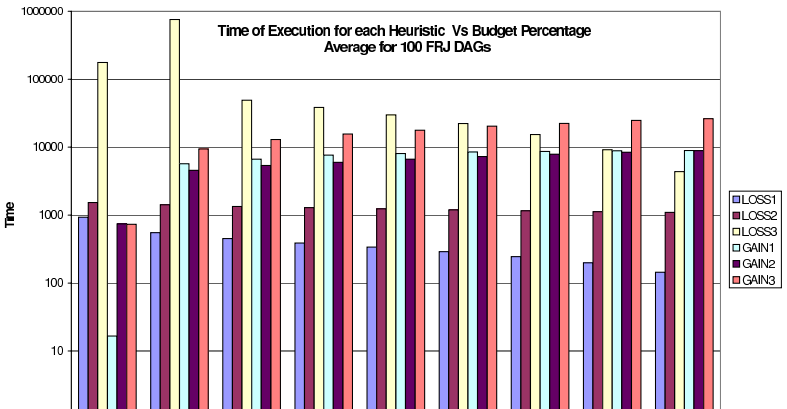

APD of each Heuristic Vs Percentage of Budget
100 FRJ DAGs
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3
APD
Budget Percentage


APD of each Heuristic Vs Percentage of Budget
100 LPL DAGs
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3
APD


Time of Execution for each Heuristic Vs Budget Percentage
Average Time for 100 Random DAGs
Execution Time
Budget Percentage
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3


Time of Execution for each Heuristic Vs Budget Percentage
Average for 100 LPL DAGs
Time of Execution
Budget Percentage
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3


Time of Execution for each Heuristic Vs Budget Percentage
Average for 100 FFT DAGs
Execution Time
Budget Percentage
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3


Time of Execution for each Heuristic Vs Budget Percentage
Average for 100 FRJ DAGs
Time
LOSS1
LOSS2
LOSS3
GAIN1
GAIN2
GAIN3

Regarding the execution time, it appears that the LOSS approach takes more time as we move towards a budget close to the cost of the cheapest assignment; the opposite happens with the GAIN approach. This is correlated with the starting basis of each of the two approaches.

5 Conclusion

We have implemented an algorithm to schedule DAGs onto heterogeneous machines under budget constraints. Different variants of the algorithm were modelled and evaluated. Future work could consider other types of DAGs that correspond to workflows of interest in the Grid community (e.g., [1, 9]), could include more sophisticated models to charge for machine time (although relevant research in the context of the Grid is still in its infancy), and consider more dynamic scenarios and environments for the execution of the DAGs and the modelling of the machines.

References

1. J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy. Resource Allocation Strategies for Workflows in Grids In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*.
2. R. Buyya, D. Abramson, and S. Venugopal. The Grid Economy. In *Proceedings of the IEEE*, volume 93(3), pages 698–714, March 2005.
3. R. Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing*. PhD thesis, Monash University, Melbourne, Australia, http://www.buyya.com/thesis, April 12 2002.
4. R. Buyya, D. Abramson, and J. Giddy. An economy grid architecture for service-oriented grid computing. In *10th IEEE Heterogeneous Computing Workshop (HCW'01)*, San Fransisco, 2001.
5. A. Mandal, K. Kennedy, C. Koelbel, G. Marin, J. Mellor-Crummey, B. Liu and L. Johnsson. Scheduling Strategies for Mapping Application Workflows onto the Grid. In *IEEE International Symposium on High Performance Distributed Computing (HPDC 2005)*, 2005.
6. R. Sakellariou and H. Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *13th IEEE Heterogeneous Computing Workshop (HCW'04)*, Santa Fe, New Mexico, USA, April 2004.
7. R. Sakellariou and H. Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. In *Scientific Programming*, volume 12(4), pages 253–262, December 2004.
8. H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. In *IEEE Transactions on Parallel and Distributed Systems*, volume 13(3), pages 260–274, March 2002.
9. M. Wieczorek, R. Prodan and T. Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. In *SIGMOD Record*, volume 34(3), September 2005.
10. H. Zhao and R. Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par 2003*. Springer-Verlag, LNCS 2790, 2003.

Integration of ISS into the VIOLA Meta-scheduling Environment

Vincent Keller[1], Kevin Cristiano[2], Ralf Gruber[1], Pierre Kuonnen[2], Sergio Maffioletti[5], Nello Nellari[5], Marie-Christine Sawley[5], Michela Spada[1], Trach-Minh Tran[1], Philip Wieder[3], Oliver Wäldrich[4], and Wolfgang Ziegler[4]

[1] Ecole Polytechnique Fédérale, CH-1015 Lausanne, Switzerland
{Vincent.Keller, Ralf.Gruber, Trach-Minh.Tran, Michela.Spada}@epfl.ch
[2] Ecole d'Ingénieurs et d'Architectes, CH-1705 Fribourg, Switzerland
{Kevin.Cristiano , Pierre.Kuonen}@eif.ch
[3] Forschungszentrum Jülich GmbH, D-52425, Germany
ph.wieder@fz-juelich.de
[4] Fraunhofer Gesellschaft, St. Augustin, Germany
{Wolfgang.Ziegler, Oliver.Waeldrich}@scai.fraunhofer.de
[5] Swiss National Supercomputing Centre, CH-1015 Manno, Switzerland
{Sergio.Maffioletti , sawley, Nello.Nellari}@cscs.ch

Abstract. The authors present the integration of the Intelligent (Grid) Scheduling System into the VIOLA meta-scheduling environment which itself is based on the UNICORE Grid software. The goal of the new, integrated environment is to enable the submission of jobs to the Grid system best-suited for the application workflow. For this purpose a cost function is used that exploits information about the type of application, the characteristics of the system architectures, as well as the availabilities of the resources. This document presents an active collaboration between Ecole Polytechnique Fédérale de Lausanne (EPFL), Ecole d'Ingénieurs et d'Architectes (EIF) de Fribourg, Forschungszentrum Jülich, Fraunhofer Institute SCAI, and Swiss National Supercomputing Centre (CSCS).

1 Introduction

The UNICORE middleware has been designed and implemented in various projects world-wide, for example the German UNICORE Plus project [1], the EU projects EUROGRID [2] and UniGrids [3], or the Japanese NaReGI project [4]. A recently developed extension to UNICORE, the VIOLA Meta-Scheduling Service, strongly increases its functionalities by adding capabilities needed to schedule arbitrary resources in a co-ordinated fashion. This meta-scheduling environment provides the software basis for the VIOLA testbed [5] and offers the opportunity to include proprietary scheduling solutions. The Intelligent (Grid) Scheduling System (ISS) [6] is such a scheduling system. It uses historical runtime data of an application to schedule a well suited computational resources for execution based on the performance requirements of the user. The goal of the work presented here is to integrate the ISS into the meta-scheduling environment to realise a Grid system satisfying the requirements of the SwissGRID.

The Intelligent Scheduling System will add a data repository, a broker and an information service to the resulting Grid system. The scheduling algorithm used to calculate the best-suited system is based on a cost function that takes the data collected during previous executions into account describing inter alia the type of the application, its performance on the different machines in the Grid, and their availability.

In the following section, the functions of UNICORE and the Meta-Scheduling Service are shortly presented. Then, the ISS model is introduced followed by a description of the overall architecture which illustrates the integration of the ISS concept into the VIOLA environment (Sections 3 and 4). Section 5 then outlines the processes that will be executed to schedule application workflows in the meta-scheduling environment. Subsequent to the generic process description an ORB5 application example that runs on machines with over 1000 processors is discussed in Section 6. We conclude this document with a summary and a brief outlook on future work.

2 UNICORE and the Meta-scheduling Service

The basic Grid environment we use for our work comprises the UNICORE Grid system and the Meta-Scheduling Service developed in the VIOLA project. It is not the purpose of this document to introduce these systems in detail, but a short characterisation of both is given in the following two sections. Descriptions of UNICORE's models and components can be found in other publications [1],[7], respective in publications covering the Meta-Scheduling Service [8], [9], [10].

2.1 UNICORE

A workflow is in general submitted to a UNICORE Grid via the UNICORE Client (see Fig. 1) which provides means to construct, monitor and control workflows. In addition the client offers extension capabilities through a plug-in interface, which has for example been used to integrate the Meta-Scheduling Service into the UNICORE Grid system. The workflow then passes the security Gateway and is mapped to the site-specific characteristics at the UNICORE Server before being transferred to the local scheduler.

The concept of resource virtualisation manifests itself in UNICORE's Virtual Site (Vsite) that comprises a set of resources. These resources must have direct access to each other, a uniform user mapping, and they are generally under the same administrative control. A set of Vsites is represented by a UNICORE Site (Usite) that offers a single access point (a unique address and port) to the resources of usually one institution.

2.2 Meta-Scheduling Service

The meta-scheduler is implemented as a Web Service receiving a list of resources preselected by a resource selection service (a broker for example, or a user) and

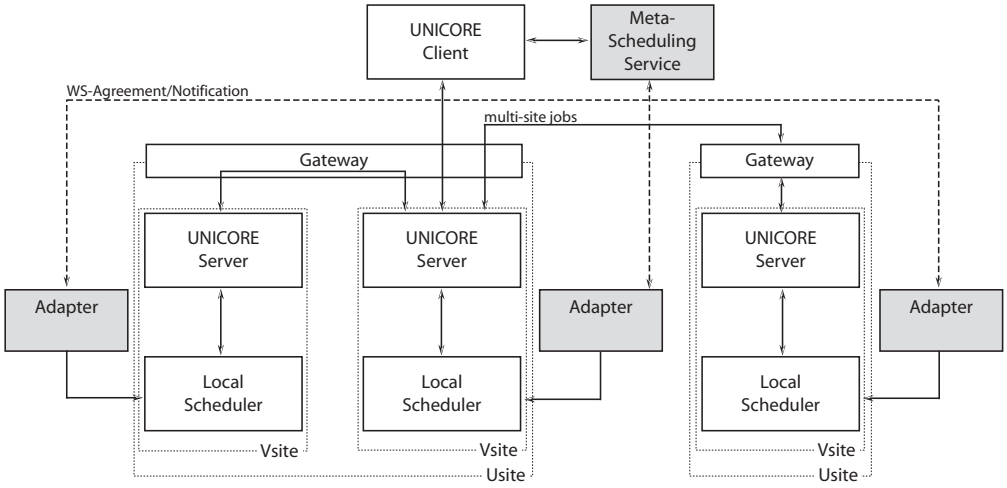


Fig. 1. Architecture of the VIOLA Meta-scheduling Environment

returning reservations for some or all of these resources. To achieve this, the Meta-Scheduling Service first queries selected local scheduling systems for the availability of these resources and then negotiates the reservations across all local scheduling systems. In the particular case of the meta-scheduling environment the local schedulers are contacted via an adapter which provides a generic interface to these schedulers. Through this process the Meta-Scheduling Service supports scheduling of arbitrary resources or services for dedicated times. It offers on one hand the support for workflows where the agreements about resource or service usage (aka reservations) of consecutive parts should be made in advance to avoid delay during the execution of the workflow. On the other hand the Meta-Scheduling Service also supports co-allocation of resources or services in case it is required to run a parallel distributed application which needs several resources with probably different characteristics at the same time. The meta-scheduler may be steered directly by a user through a command-line interface or by Grid middleware components like the UNICORE client through its SOAP interface (see Fig. 1). The resulting reservations are implemented using the WS-Agreement specification [11].

3 Intelligent Scheduling System Model

The main objective of the Intelligent GRID Scheduling System (ISS) project [6] is to provide a middleware infrastructure allowing optimal positioning and scheduling of real life applications in a computational GRID. According to data collected on the machines in the GRID, on the behaviour of the applications, and on the performance requirements demanded by the user, a well suited computational resource is detected and allocated to execute the application. The monitoring information collected during execution is put into a database and reused for the next resource allocation decision. In addition to providing scheduling informa-

tion, the collected data allows to detect overloaded resources and to pin-point inefficient applications that could be further optimised.

3.1 Application types

The Intelligent Scheduling System model is based on the following application type system:

- **Single Processor Applications** These applications do not need any internode communication. They may benefit from backfilling strategies.
- **Embarrassingly parallel applications** This kind of applications requires a client-server concept. The internode communication network is not important. Seti@Home is an example of an embarrassingly parallel application for which data is sent over the Web.
- **Point-to-point applications** Point-to-point communications typically appear in finite element or finite volume methods when a huge 3D domain is decomposed in sub-domains and an explicit time stepping method or an iterative matrix solver is applied. If the number of processors grows with the problem size, and the size of a sub-domain is fixed, the local problem size is fix. Hence, that kind of applications can run well on a cluster with a relatively slow and cost-effective communication network that scales with the number of processors.
- **Multicast communications applications** The parallel 3D FFT algorithm is a typical example of an application that is dominated by multicast operations. The internode communication increases with the number of processors. Such an application needs a faster switched network such as Myrinet, Quadrics, or Infiniband. If thousands of processors are needed, special-purpose machines such as RedStorm or BlueGene might be required.
- **Multi components applications** Such applications consist of well-separable components, each one being a parallel job with little inter-component interaction. The different components can be submitted to different machines. An example is presented in [13].

The ISS concept is straight-forward: if a scheduler is able to differentiate between the types of applications presented above, it can decide where to run an application. For this purpose the so-called Γ model has been developed which is described in the following.

3.2 The Γ model

In the Γ model described in [12], it is supposed that each component of the application is ideally parallelized, i.e. each task of a component takes the same CPU and communication times.

The most important parameter Γ is a measure of the ratio of the computation over the communication times of each component. An application component adapted parallel machine should have a $\Gamma > 1$. Specifically, $\Gamma = 1$ means that communication and computation times are equal.

4 Resulting Grid Middleware Architecture

The overall architecture of the ISS integration into the meta-scheduling environment is depicted in Fig. 2 and the different modules and services are presented in this section. Please note that it is assumed that the executables of the application components already exist before execution.

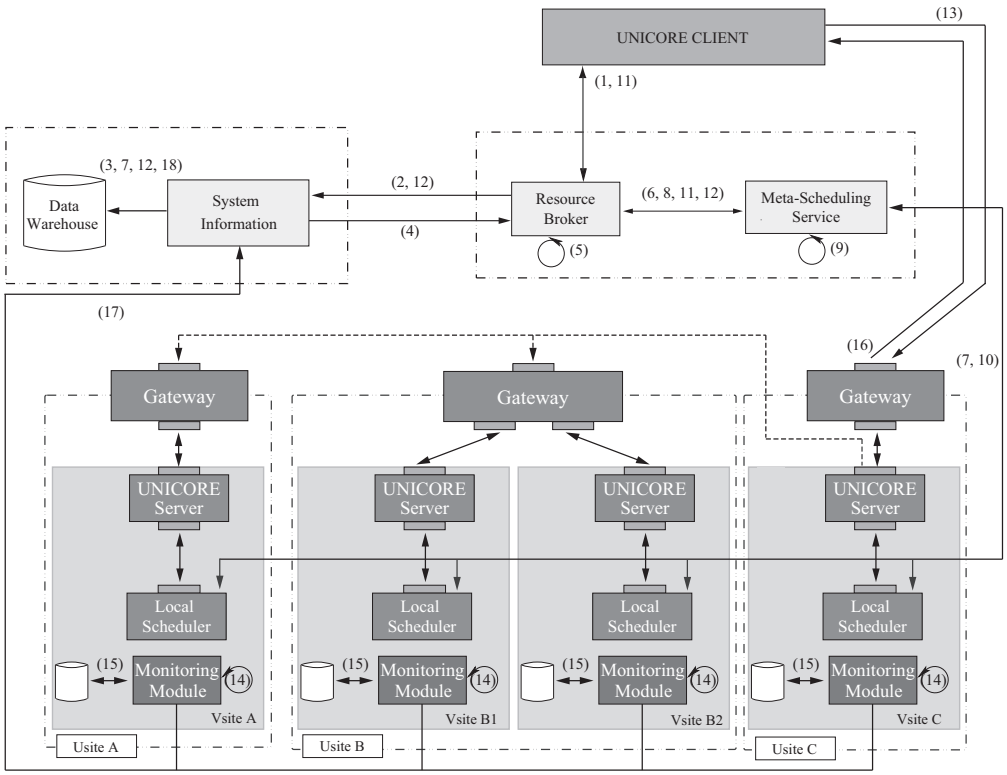


Fig. 2. Integration of ISS into the meta-scheduling environment.

4.1 Meta-Scheduling Service

The Meta-Scheduling Service (MSS) receives from the Resource Broker (RB) the resource requirements of an application, namely the number of nodes (or a set of numbers of nodes in case of a distributed parallel application) and the planned or estimated execution time. The MSS queries for the availability of known resources. MMS selects a suited machine by optimizing an objective function composed by the Γ model (described above) and the evaluation of costs. The MSS tries to reserve the proposed resource(s) for the job. The result of the reservation is sent back to the RB to check whether the final reservation matches the initial request. In case of a mismatch the reservation process will be re-iterated.

4.2 Resource Broker

The Resource Broker receives requests from the UNICORE Client, collects the necessary information to choose the set of acceptable machines in the prologue phase.

4.3 Data Warehouse

We assume that information about application components exists at the Data Warehouse (DW) module. It is also assumed that at least one executable of all the application components exists.

The DW is the database that keeps all the information related to the application components, to the resources, to the services provided by the Vsites, to monitoring, and to other parameters potentially used to calculate the cost function.

Specifically, the Data Warehouse module contains the following information:

1. **Resources** Application independent hardware quantities.
2. **Services** The services a machines provides (software, libraries installed, etc.).
3. **Monitoring** Application dependent hardware quantities collected after each execution.
4. **Applications** Γ model quantities computed after each execution of an application component.
5. **Other** Other information needed in the cost function such as cost of one hour engineering time.

4.4 System Information

The System Information (SI) module manages the DW, accesses the Vsite-specific UNICORE information service periodically to update the static data in the DW, receives data from the Monitoring Module (MM) and the MSS, and interacts with the RB.

4.5 Monitoring Module

The Monitoring Module collects the application relevant data per Vsite during the runtime of an application. Specifically, it collects dynamic resource information (like CPU usage, network packets number and size, memory usage, etc..), and sends it to the SI (It is an extension of the present TSI).

5 Detailed Scheduling Scenario

Fig. 2 also shows the processes which are executed after a workflow is submitted to the Grid system we have developed. The 18 steps are broken down into three different phases: prologue, scheduling/execution, and epilogue.

First phase: Prologue

(1) The user submits a workflow to the RB through the UNICORE Client.
(2) The RB asks SI for systems able to run each workflow components (in terms of cost, amount of memory, parallel paradigme, etc...)
(3) The SI request the information from the DW
(4) The SI sends the information back to the RB.
(5) According to the information obtained in (3) the RB selects resources that might be used to run the job.
(6) The RB sends the list of resources together with further information (like number of nodes, expected run-time, etc.) and a user certificate to the MSS.
(7) The MSS collects information across all pre-selected resources about availability (e.g. of the compute nodes or of necessary licenses), user-related policies (like access rights), and cost-related parameters.
(8) The MSS notifies the RB about the completion of the prologue phase.

Second phase: Optimal Scheduling and execution

(9) The MSS can now choose among a number of acceptable machines that could execute the workflow. To select a well suited one, it uses consolidated information about each Vsite, e.g. the number of nodes, the memory size per node M_{Vsite}, or the cost for 1 CPU hour per node. The MSS then calculates the cost function to find a well suited resource for the execution of the workflow. Knowing the amount of memory needed by the application, M_a, the MSS can determine the number of nodes P ($P > M_a/M_{Vsite}$) and compute the total time T:
$Total\ time\ T\ =\ Waiting\ Time\ T_w\ +\ Computation\ Time\ T_c$
needed in the cost function. The MSS chooses the machine(s).
(10) The MSS contacts the local scheduling system(s) of the selected resource(s) and tries to obtain a reservation.
(11) If the reservation is confirmed the MSS creates an agreement, sends it to the UNICORE Client via the RB.
(12) The MSS then forwards the decision made in (9) via the RB to the SI which puts the data into the DW.
(13) The UNICORE Client creates the workflow based on the agreement and submits it to the UNICORE Gateway. Subsequent parts of the workflow are handled by the UNICORE Server of the submission Usite.
(14) During the workflow execution, application characteristics, such as CPU usage, network usage, number and size of MPI and NFS messages, and the amount of memory used, are collected by the MM.
(15) The MM stores the information in a local database.
(16) The result of the computation is sent back to the UNICORE Client.

Third phase: Epilogue

(17) Once the workflow execution has finished, the MM sends data stored during the computation to the SI.
(18) The SI computes the Γ model parameters and writes the relevant data into the DW.

The user only has to submit the workflow, the subsequent steps including the selection of well suited resource(s) are transparent to him. Only if an application is executed for the first time, the user has to give some basic information since no application-specific data is present in the DW.

There is a number of uncertainties in the computation of the cost model. The parameters used in the cost function are those that were measured in a previous execution of the same application. However, this previous execution could have used a different input pattern. Additionally, the information queried from the different resources by the MSS is based on data that has been provided by the application (or the user) before the actual execution and may therefore be rather imprecise. In future, by using ISS, such estimations could be improved.

During the epilogue phase data is also collected for statistical purpose. This data can provide information about reasons for a resource's utilisation or a user's satisfaction. If this is bad for a certain HPC resource, for instance because of overfilled waiting queues, other machines of this type should be purchased. If a resource is rarely used it either has a special architecture or the cost charged using it is too high. In the latter case one option would be to adapt the price.

6 Application Example: Submission of ORB5

Let us follow the data flow of the real life plasma physics application ORB5 that runs on parallel machines with over 1000 processors. ORB5 is a particle in cell code. The 3D domain is discretised in $N_1 x N_2 x N_3$ mesh cells in which move p charged particles. These particles deposit their charges in the local cells. Maxwell's equation for the electric field is then solved with the charge density distribution as source term. The electric field accelerates the particles during a short time and the process repeats with the new charge density distribution. As a test case, $N_1 = N_2 = 128$, $N_3 = 64$, $p = 2'000'000$, and the number of time steps is $t = 100$. These values form the ORB5 input file.

Two commodity clusters at EPFL form our test Grid, one having 132 single processor nodes interconnected with a full Fast Ethernet switch (Pleiades), the other has 160 two processor nodes interconnected with a Myrinet network (Mizar).

The different steps in decision to which machine the ORB5 application is submitted are:

(1) The ORB5 execution script and input file are submitted to the RB through a UniCORE client.
(2) The RB requests information on ORB5 from the SI.
(3) The SI selects the information from the DW (memory needed 100 GB, $\Gamma = 1.5$ for Pleiades, $\Gamma = 20$ for Mizar, 1 hour engineering time cost Sfr. 200.-, 8 hours a day).

(4) SI sends back to RB the information.
(5) RB selects Mizar and Pleiades.
(6) RB sends the information on ORB5 to MSS
(7) MSS collects machine information from Pleiades and Mizar:
  - **Pleiades:** 132 nodes, 2 GB per node, SFr. 0.50 per node*h, 2400 h*node job limit, availability table (1 day for 64 nodes), user is authorised, executable ORB5 exist.
  - **Mizar:** 160 nodes, 4 GB per node, SFr. 2.50 per node*h, 32 nodes job limit, availability table (1 hour for 32 nodes), user is authorised, executable ORB5 exist.

(8) Prologue is finished.
(9) MSS computes the cost function values using the estimated execution time of 1 day:
  - **Pleiades:** Total costs = Computing costs (24*64*0.5=SFr. 768.-) + Waiting time ((1+1)*8*200=SFr. 3200.-) = SFR 3968.-
  - **Mizar:** Total costs = Computing costs (24*32*2.5=SFr.1920.-) + Waiting time ((1+8)*200=SFr. 1800.-) = SFR 3720.-

  MSS decides to submit to Mizar.
(10) MSS requests the reservation of 32 nodes for 24 hours from the local scheduling system of Mizar.
(11) If the reservation is confirmed the MSS creates the agreement, sends it to UC. Otherwise the broker is notified and the selection process will start again.
(12) MSS sends the decision to use Mizar to SI via the RB.
(13) UC submits the ORB5 job to the UNICORE gateway.
(14) Once the job is executed on the 32 nodes the execution data is collected by MM.
(15) MM sends execution data to local database.
(16) Results of job are sent to UC.
(17) MM sends the job execution data stored in the local database to the SI.
(18) SI computes Γ model parameters (e.g. $\Gamma = 18.7$, $M = 87$ GB, Computing time=21h 32') and stores them into DW.

7 Conclusion

The ISS integration into the VIOLA Meta-scheduling environment is part of the SwissGRID initiative and will be realised in a co-operation between CoreGRID partners. It is planned to install the resulting Grid middleware by the end of 2007 to guide job submission to all HPC machines in Switzerland.

8 Acknowledgements

Some of the work reported in this paper is funded by the German Federal Ministry of Education and Research through the VIOLA project under grant #123456. This paper also includes work carried out jointly within the CoreGRID Network of Excellence funded by the European Commission's IST programme under grant #004265.

References

1. D. Erwin (ed.), *UNICORE plus final report – uniform interface to computing resource*, Forschungszentrum Jülich, ISBN 3-00-011592-7, 2003.
2. The EUROGRID project, website, 2005. Online: http://www.eurogrid.org/.
3. The UniGrids Project, website, 2005. Online: http://www.unigrids.org/.
4. The National Research Grid Initiative (NaReGI), website, 2005. Online: http://www.naregi.org/index_e.html
5. VIOLA – Vertically Integrated Optical Testbed for Large Application in DFN, website, 2005. Online: http://www.viola-testbed.de/.
6. R. Gruber, V. Keller, P. Kuonen, M.-Ch. Sawley, B. Schaeli, A. Tolou, M. Torruella, and T.-M. Tran, Intelligent Grid Scheduling System, In *Proc. of Conference on Parallel Processing and Applied Mathematics PPAM 2005*, Poznan, Poland, 2005, to appear.
7. A. Streit, D. Erwin, Th. Lippert, D. Mallmann, R. Menday, M. Rambadt, M. Riedel, M. Romberg, B. Schuller, and Ph. Wieder, UNICORE - From Project Results to Production Grids, L. Grandinetti (ed.), *Grid Computing and New Frontiers of High Performance Processing*, Elsevier, 2005, to be published. Pre-print available at: http://arxiv.org/pdf/cs.DC/0502090.
8. G. Quecke and W. Ziegler, MeSch – An Approach to Resource Management in a Distributed Environment, In *Proc. of 1st IEEE/ACM International Workshop on Grid Computing (Grid 2000)*. Volume 1971 of Lecture Notes in Computer Science, pages 47-54, Springer, 2000.
9. A. Streit, O. Wäldrich, Ph. Wieder, and W. Ziegler, On Scheduling in UNICORE - Extending the Web Services Agreement based Resource Management Framework, In *Proc. of Parallel Computing 2005 (ParCo2005)*, Malaga, Spain, 2005, to appear.
10. O. Wäldrich, Ph. Wieder, and W. Ziegler, A Meta-scheduling Service for Co-allocating Arbitrary Types of Resource, In *Proc. of Conference on Parallel Processing and Applied Mathematics PPAM 2005*, Poznan, Poland, 2005, to appear.
11. A. Andrieux et. al., Web Services Agreement Specification, July, 2005. Online: https://forge.gridforum.org/projects/graap-wg/document/WS-Agreement-Specification/en/16.
12. Ralf Gruber, Pieter Volgers, Alessandro De Vita, Massimiliano Stengel, and Trach-Minh Tran, Parameterisation to tailor commodity clusters to applications, *Future Generation Comp. Syst.*, 19(1), pp.111-120, 2003.
13. P. Manneback, G. Bergère, N. Emad, R. Gruber, V. Keller, P. Kuonen, S. Noël, and S. Petiton, Towards a scheduling policy for hybrid methods on computational Grids, submitted to CoreGRID Integrated Research in Grid Computing workshop Pisa, 28 - 30 November, 2005.

Synthetic Grid Workloads with Ibis, KOALA, and GrenchMark

Alexandru Iosup[1], Jason Maassen[2], Rob van Nieuwpoort[2], and Dick H.J. Epema[1]

[1] Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, The Netherlands,
{A.Iosup,D.H.J.Epema}@ewi.tudelft.nl

[2] Department of Computer Science,
Vrije Universiteit, Amsterdam, The Netherlands
{Jason,Rob}@cs.vu.nl

Abstract. Grid computing is becoming the natural way to aggregate and share large sets of heterogeneous resources. However, grid development and acceptance hinge on proving that grids reliably support real applications. A step in this direction is to combine several grid components into a demonstration and testing framework. This paper presents such an integration effort, in which three research prototypes, namely a grid application development toolkit (Ibis), a grid scheduler capable of co-allocating resources (KOALA), and a synthetic grid workload generator (GRENCHMARK), are used to generate and run workloads comprising well-established and new grid applications on our DAS multi-cluster testbed.

Keywords: Grid, performance evaluation, synthetic workloads.

1 Introduction

Grid computing's long term goal is to become the natural way to share heterogeneous resources, and to aggregate them into virtual platforms, used by multiple organizations and independent users. With the grid infrastructure starting to meet the requirements of such an ambitious goal [2], the current evolution of grids hinges on proving that it can run real applications, from traditional sequential and parallel applications to new, grid-only, applications. As a consequence, there is a clear need for generating and running workloads comprising of grid applications for demonstration and testing purposes.

A significant number of projects have tried to tackle this problem from different angles: attempting to produce a representative set of grid applications like the NAS Grid Benchmarks [7], creating synthetic applications that can assess the status of grid services like the GRASP project [4], and creating tools for launching benchmarks and reporting results like the GridBench project [14].

This work addresses the problem of generating and running synthetic grid workloads, by integrating the results of three research projects coming from

CoreGRID partners, namely the grid application development toolkit Ibis [15], the grid scheduler KOALA [11], and the synthetic grid workload generator and submitter GRENCHMARK. Ibis is being developed at VU Amsterdam[3] and provides a set of generic Java-based grid applications. KOALA is being developed at TU Delft[4] and allows running generic grid applications. Finally, GRENCHMARK is being developed at TU Delft[5] and is able to generate workloads comprising typical grid applications, and to submit them to arbitrary grid environments.

2 A Case for Synthetic Grid Workloads

There are three ways of evaluating the performance of a grid system: analytical modeling, simulation, and experimental testing. This section presents the benefits and drawbacks of each of the three, and argues for evaluating the performance of grid systems using synthetic workloads, one of the two possible approaches for experimental testing.

2.1 Analytical Modeling and Simulations

Analytical modeling is a traditional method for gaining insights into the performance of computing systems. Analytical modeling may simplify *what-if* analysis, for changes in the system, in the middleware, or in the applications. However, the sheer size of grids and their heterogeneity make realistic analytical modeling hardly tractable.

Simulations may handle complex situations, sometimes very close to the real system. Furthermore, simulations allow the *replay* of real situations, greatly facilitating the discovery of appropriate solutions. However, simulated system size and diversity raises questions on the representativeness of simulating grids. Moreover, nondeterminism and other forms of hidden dynamic behavior of grids make the simulation approach even less suitable.

2.2 Experimental Testing

There are two ways to experimentally assess the performance of grid systems: *benchmarking* and *using synthetic grid workloads*. Note that currently existing grids prevent the use of traces of real grid workloads: the infrastructure changes too fast, leading to incompatible resource requests when re-running old traces.

Benchmarking is typically used to understand the quantitative aspects of running grid applications and to make results readily available for comparison. Benchmarks comprise a set applications representative for a class of systems, and a set of rules for running the applications as a synthetic system workload. Therefore, a benchmark is a single instance of a synthetic workload.

[3] Ibis is available from `http://www.cs.vu.nl/ibis/`.
[4] KOALA is available from `http://www.st.ewi.tudelft.nl/koala/`.
[5] GRENCHMARK is available from `http://grenchmark.st.ewi.tudelft.nl/`.

Benchmarks present severe limitations, when compared to synthetic grid workloads generation. They have to be developed under the auspices of an important number of (typically competing) entities, and can only include well-studied applications. Putting aside the considerable amounts of time and resources needed for these tasks, the main problem is that grid applications are starting to develop just now, typically at the same time with the infrastructure [12], thus limiting the availability of truly representative applications for inclusion in standard benchmarks. Other limitations in using benchmarks for more than raw performance evaluation are:

- Benchmarking results are valid only for workloads truly represented by the benchmark's set of applications; moreover, the number of applications typically included in benchmarks [7, 14] is typically small, limiting even more the scope of benchmarks;
- Benchmarks include mixes of applications representative at a certain moment of time, and are notoriously resistant to include new applications; thus, benchmarks cannot respond to the changing requirements of developing infrastructures, such as grids;
- Benchmarks measure only one particular system characteristic (low-level benchmarks), or a mix of characteristics (high-level benchmarks), but not both;

An extensible framework for *generating and submitting synthetic grid workloads* uses applications representative for today's grids, and fosters the addition of future grid applications. This approach can help overcome the aforementioned limitations of benchmarks. First, it offers better flexibility in choosing the starting applications set, when compared to benchmarks. Second, applications can be included in generated workloads, even when they are in a debug or test phase. Third, the workload generation can be easily parameterized, to allow for the evaluation of one or a mix of system characteristics.

2.3 Grid Applications Types

From the point of view of a grid scheduler, we identify two types of applications that can run in grids, and may be therefore included in synthetic grid workloads.

Unitary applications This category includes single, unitary, applications. At most the job programming model must be taken into account when running in grids (e.g., launching a name server before launching an Ibis job). Typical examples include sequential and parallel (e.g., MPI, Java RMI, Ibis) applications. The tasks composing a unitary application, for instance in a parallel application, can interact with each other.

Composite applications This category includes applications composed of several unitary or composite applications. The grid scheduler needs to take into account issues like task inter-dependencies, advanced reservation and extended fault-tolerance, besides the components' job programming model. Typical examples include parameter sweeps, chains of tasks, DAG-based applications, and even generic graphs.

2.4 Purposes of Synthetic Grid Workloads

We distinguish three reasons for using synthetic grid workloads.

System design and procurement Grid architectures offer many alternatives to their designers, in the form of hardware, of operating software, of middleware (e.g., a large variety of schedulers), and of software libraries. When a new system is replacing an old one, running a synthetic workload can show whether the new configuration performs according to the expectations, before the system becomes available to users. The same procedure may be used for assessing the performance of various systems, in the selection phase of the procurement process.

Functionality testing and system tuning Due to the inherent heterogeneity of the grids, complicated tasks may fail in various ways, for example due to misconfiguration or unavailability of required grid middleware. Running synthetic workloads, which use the middleware in ways similar to the real application, helps testing the functionality of the grids and detecting many of the existing problems.

Performance testing of grid applications With grid applications being more and more oriented towards services [9] or components [8], early performance testing is not only possible, but also required. The production cycle of traditional parallel and distributed applications must include early testing and profiling. These requirements can be satisfied with a synthetic workload generator and submitter.

3 An Extensible Framework for Grid Synthetic Workloads

This section presents an extensible framework for generating and submitting synthetic grid workloads. The first implementation of the framework integrates two research prototypes, namely a grid application development toolkit (Ibis), and a synthetic grid workload generator (GrenchMark).

3.1 Ibis: Grid Applications

Ibis is a grid programming environment offering the user efficient execution and communication [5], and the flexibility to run on dynamically changing sets of heterogeneous processors and networks.

The Ibis distribution package comes with over 30 working applications, in the areas of physical simulations, parallel rendering, computational mathematics, state space search, bioinformatics, prime numbers factorization, data compression, cellular automata, grid methods, optimization, and generic problem solving. The Ibis applications closely resemble real-life parallel applications, as they cover a wide-range of computation/communication ratios, have different communication patterns and memory requirements, and are parameterized. Many

of the Ibis applications report detailed performance results. Last but not least, all the Ibis applications have been thoroughly described and tested in various grids [5, 15]. They work on various numbers of machines, and have automatic fault tolerance and migration features, thus responding to the requirements of dynamic environments such as grids. For a complete list of publications, please visit `http://www.cs.vu.nl/ibis`. Therefore, the Ibis applications are representative for grid applications written in Java, and can be easily included in synthetic grid workloads.

3.2 GrenchMark: Synthetic Grid Workloads

GrenchMark is a synthetic grid workload generator and submitter. It is *extensible*, in that it allows new types of grid applications to be included in the workload generation, *parameterizable*, as it allows the user to parameterize the workloads generation and submission, and *portable*, as its reference implementation is written in `Python`.

The workload generator is based on the concepts of *unit generators* and of job description files (JDF) *printers*. The *unit generators* produce detailed descriptions on running a set of applications (*workload unit*), according to the workload description provided by the user. In principle, there is one unit for each type of supported application type. The *printers* take the generated workload units and create job description files suitable for grid submission. In this way, multiple unit generators can be coupled to produce a workload that can be submitted to any grid resource manager, as long as the resource manager supports that type of applications.

The grid applications currently supported by GrenchMark are sequential jobs, jobs which use MPI, and Ibis jobs. We use the Ibis applications included in the default Ibis distribution (see Section 3.1). We have also implemented three *synthetic applications*: `sser`, a sequential application with parameterizable computation and memory requirements, `sserio`, a sequential application with parameterizable computation and I/O requirements, and `smpi1`, an MPI application with parameterizable computation, communication, memory, and I/O requirements. Currently, GrenchMark can submit jobs to Koala, Globus GRAM, and Condor.

The workload generation is also dependent on the applications inter-arrival time [6]. Peak job arrival rates for a grid system can also be modeled using well-known statistical distributions [6, 10]. Besides the Poisson distribution, used traditionally in queue-based systems simulation, modeling could rely on uniform, normal, exponential and hyper-exponential, Weibull, log normal, and gamma distributions. All these distributions are supported by the GrenchMark generator.

The workload submitter generates detailed reports of the submission process. The reports include all job submission commands, the turnaround time of each job, including the grid overhead, the total turnaround time of the workload, and various statistical information.

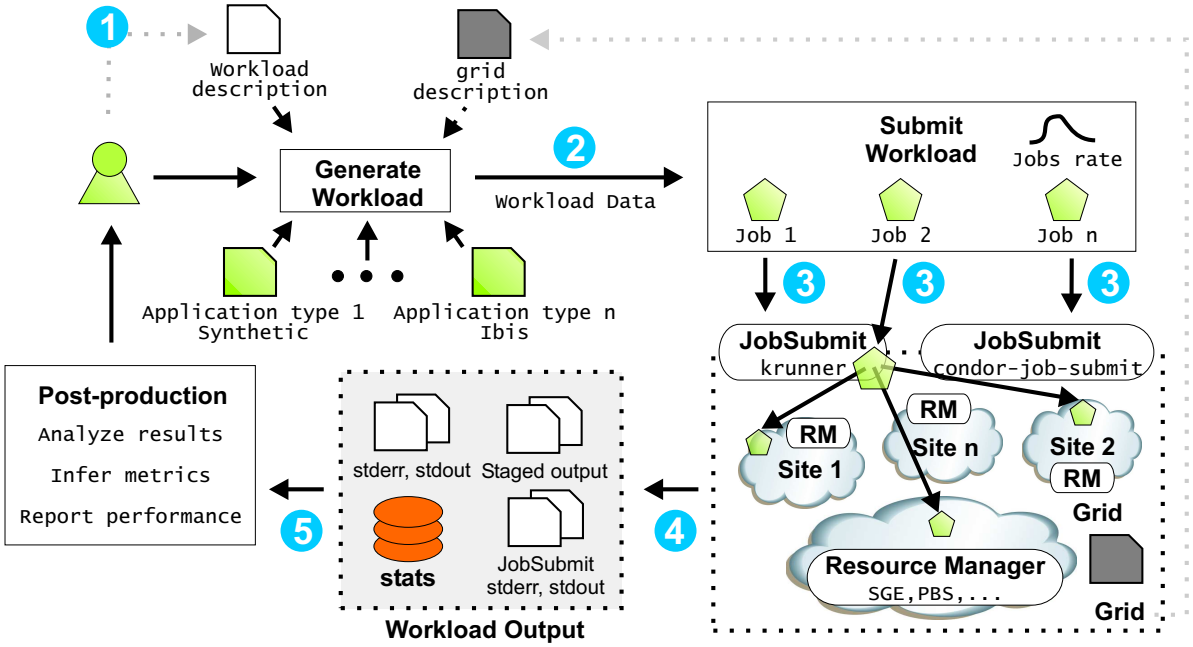


Fig. 1. The GrenchMark process.

3.3 Using the Framework

Figure 1 depicts the typical usage of our framework. First, the user describes the workload to be generated, as a formatted text file (1). Based on the user description, on the known application types, and on information about the grid sites, a workload is then generated by GrenchMark (2), and submitted to the grid (3). The grid environment is responsible for executing the jobs and returning their results (4). The results include not only job outcomes, but also detailed submission reports. Finally, the user processes all results in a post-production step (5).

4 A Concrete Case: Synthetic Workloads for the DAS

This section presents a concrete case for our framework: generating and running synthetic workloads on the DAS [1]. The Ibis applications were combined with the synthetic applications, to create a pool of over 35 grid applications. The GrenchMark tools were used to generate and launch the synthetic workloads.

4.1 Koala: Scheduling Grid Applications

A key part of the experimental infrastructure is the Koala [11] grid scheduler. To the author's knowledge, Koala is the only fault-tolerant, well-tested, and deployed grid scheduler that provides support for *co-allocated* jobs, that is, it can simultaneously allocate resources in multiple grid sites to single applications which consist of multiple components. Koala was used to submit the generated workloads to the DAS multi-cluster. Its excellent reporting capabilities were also used for evaluating the jobs execution results.

Workload	Applications types	# of Jobs	# of CPUs	# of Components	Component Size	Success Rate
`gmark1`	synthetic, sequential	100	1	1	1	97%
`gmark+`	synthetic, seq. & MPI	100	1-128	1-15	1-32	81%
`ibis1`	N Queens, Ibis	100	2-16	1-8	2-16	56%
`ibis+`	various, Ibis	100	2-32	1-8	2-16	53%
`wl+all`	all types	100	1-32	1-8	1-32	90%

Table 1. The experimental workloads. As the DAS has only 5 sites; jobs with more than 5 components will have several components running at the same site.

```
# File-type: text/wl-spec
#ID Jobs Type   SiteType Total SiteInfo ArrivalTimeDistr   OtherInfo
?   25   sser   single     1      *:?    Poisson(120s)      StartAt=0s
?   25   sserio single     1      *:?    Poisson(120s)      StartAt=60s
?   25   smpi1  single     1      *:?    Poisson(120s)      StartAt=30s,ExternalFile=smpi1.xin
?   25   smpi1  single     1      *:?    Poisson(120s)      StartAt=90s,ExternalFile=smpi2.xin
```

Fig. 2. A GrenchMark workload description example.

For co-allocated jobs, Koala gives the user the option to specify the actual execution sites, i.e., the clusters where job components should run. Koala supports *fixed* jobs, for which users fully specify the execution sites, *non-fixed* jobs, for which the user does not specify the execution sites, leaving instead Koala to select the best sites, and *semi-fixed* jobs, which are a mix of the previous two. Koala may schedule different components of a non-fixed or of a semi-fixed job onto the same site. We used this feature heavily for the Ibis and synthetic MPI applications.

4.2 Workload Generation

Table 1 shows the structure of the five generated workloads, each comprising 100 jobs. To satisfy typical grid situations, jobs request resources from 1 to 15 sites. For parallel jobs, there is a preference for 2 and 4 sites. Site requests are either precise (specifying the full name of a grid site) or non-specified (leaving the scheduler to decide). For multi-site jobs, components occupy between 2 and 32 processors, with a preference for 2, 4, and 16 processors. We used combinations of parameters that would keep the run-time of the applications under 30 minutes, under optimal conditions. Each job requests resources for a time below 15 minutes. Various inter-arrival time distributions are used, but the submission time of the last job of any workload is kept under two hours.

Figure 2 shows the workload description for generating the `gmark+` test, comprising 100 jobs of four different types. The first two lines are comments. The next two lines are used to generate sequential jobs of types `sser` and `sserio`, with default parameters. The final two lines are used to generate MPI jobs of type `smpi1`, with parameters specified in external files `smpi1.xin` and `smpi2.xin`. All four job types assume an arrival process with Poisson distribution, with a average rate of 1 job every 120 seconds. The first job of each type starts at a time specified in the workload description with the help of the `StartAt` tag.

Job name	Job type	Turnaround [s] Avg.	Min	Max	Runtime [s] Avg.	Min	Max	Run	Run+ Success
`sser`	sequential	129	16	926	44	1	588	100%	97%
`smpi1`	MPI	332	21	1078	110	1	332	80%	85%
`N Queens`	Ibis	99	15	1835	31	1	201	66%	85%

Table 2. A summary of time and run/success percentages for different job types.

4.3 The Workload Submission

GrenchMark was used to submit the workloads. Each workload was submitted in the normal DAS working environment, thus being influenced by the background load generated by other DAS users. Some jobs could not finish in the time for which they requested resources, and were stopped automatically by the Koala scheduler. This situation corresponds to users under-estimating applications' runtimes. Each workload ran between the submission start time and 20 minutes after the submission of the last job. Thus, some jobs did not run, as not enough free resources were available during the time between their submission and the end of the workload run. This situation is typical for real working environments, and being able to run and stop the workload according to the user specifications shows some of the capabilities of GrenchMark.

5 Experimental results

This section presents an overview of the experimental results, and shows that workloads generated with GrenchMark can cover in practice a wide-range of run characteristics.

5.1 Performance Results

Table 1 shows the success rate for all five workloads (column *Success Rate*). A successful job is a job that gets its resources, runs, finishes, and returns all results within the time allowed for the workload. The lower performance of Ibis jobs (workload `ibis+`) when compared to all the others, is caused by the fact that the system was very busy at the time of testing, making the resource allocation particularly difficult. This situation cannot be prevented in large-scale environments, and cannot be addressed without special resource reservation rights.

The turnaround time of an application can vary greatly (see Table 2), due to different parameter settings, or to varying system load. The variations in the application runtimes are due to different parameter settings.

As expected, the percentage of the applications that are actually run (Table 2, column *Run*) depends heavily on the job size and system load. The success rate of jobs that *did* run shows little variation (Table 2, column *Run+Success*). The ability of GrenchMark to report percentages such as these enables future work on comparing of the success rate of co-allocated jobs, vs. single-site jobs.

5.2 Dealing With Errors

Using the combined GrenchMark and Koala reports, it was easy to identify errors at various levels in the submission and execution environment: the user, the scheduler, the local and the remote resource, and the application environment levels. For a better description of the error levels, and for a discussion about the difficulty of trapping and understanding errors, we refer the reader to the work of Thain and Livny [13].

We were able to identify bottlenecks in the grid infrastructure, and in particular in Koala, which was one of our goals. For example, we found that for large jobs in a busy system, the percentage of unsuccessful jobs increases dramatically. The reason is twofold. First, using a single machine to submit jobs (a typical grid usage scenario) incurs a high level of memory occupancy, especially with many jobs waiting for the needed resources. A possible solution is to allow a single Koala job submitter to support multiple job submissions. Second, there are cases when jobs attempt to claim the resources allocated by the scheduler, but fail to do so, for instance because a local request leads to resources being claimed by another user (scheduling-claiming atomicity problem). These jobs should not be re-scheduled immediately, or this could lead to a high occupancy of the system resources. A possible solution is to use an exponential back-off mechanism when scheduling such jobs.

6 Conclusions and Ongoing Work

This work has addressed the problem of synthetic grid workload generation and submission. We have integrated three research prototypes, namely a grid application development toolkit, Ibis, a grid metascheduler, Koala, and a synthetic grid workload generator, GrenchMark, and used them to generate and run workloads comprising well-established and new grid applications on a multi-cluster grid. We have run a large number of application instances, and presented overview results of the runs.

We are currently adding to GrenchMark the complex applications generation capabilities and an automatic results analyzer. For the future, we plan to prove the applicability of GrenchMark for specific grid performance evaluation, such as such as an evaluation of the DAS support for High-Energy Physics applications or a performance comparison of co-allocated and single site applications, to complement our previous simulation work [3].

Acknowledgements

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265). Part of this work was also carried out in the context of the Virtual Laboratory for e-Science project (`www.vl-e.nl`), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is part of the ICT innovation program of the Dutch Ministry of Economic Affairs (EZ).

We would also like to thank Hashim Mohamed and Wouter Lammers, for their work on KOALA, and Gosia Wrzesiska, Niels Drost, and Mathijs den Burger, for their work on Ibis.

References

[1] Henri E. Bal et al. The distributed ASCI supercomputer project. *Operating Systems Review*, 34(4):76–96, October 2000.

[2] F. Berman, A. Hey, and G. Fox. *Grid Computing: Making The Global Infrastructure a Reality*. Wiley Publishing House, 2003. ISBN: 0-470-85319-0.

[3] A. I. D. Bucur and D. H. J. Epema. Trace-based simulations of processor co-allocation policies in multiclusters. In *Proc. of the 12th IEEE HPDC*, pages 70–79. IEEE Computer Society, 2003.

[4] G. Chun, H. Dail, H. Casanova, and A. Snavely. Benchmark probes for grid assessment. In *IPDPS*. IEEE Computer Society, 2004.

[5] Alexandre Denis, Olivier Aumage, Rutger Hofman, Kees Verstoep, Thilo Kielmann, and Henri E. Bal. Wide-area communication for grids: An integrated solution to connectivity, performance and security problems. In *13th International Symposium on High-Performance Distributed Computing (HPDC-13)*, pages 97–106, Honolulu, Hawaii, USA, June 2004.

[6] Steve J. Chapin et al. Benchmarks and standards for the evaluation of parallel job schedulers. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 67–90. Springer-Verlag, 1999.

[7] Michael Frumkin and Rob F. Van der Wijngaart. Nas grid benchmarks: A tool for grid space exploration. *Cluster Computing*, 5(3):247–255, 2002.

[8] Vladimir Getov and Thilo Kielmann, editors. *Component Models and Systems for Grid Applications*, volume 1 of *CoreGRID seroes*. Springer Verlag, June 2004. Proceedings of the Workshop on Component Models and Systems for Grid Applications held June 26, 2004 in Saint Malo, France.

[9] M. Humphrey et al. State and events for web services: A comparison of five WS-Resource Framework and WS-Notification implementations. In *4th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, Research Triangle Park, NC, USA, July 2005.

[10] Uri Lublin and Dror G. Feitelson. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. *J. Parallel & Distributed Comput.*, 63(11):1105–1122, Nov 2003.

[11] H.H. Mohamed and D.H.J. Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *Proc. of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005)*, Cardiff, UK, May 2005.

[12] A. Snavely et al. Benchmarks for grid computing: a review of ongoing efforts and future directions. *SIGMETRICS Perform. Eval. Rev.*, 30(4):27–32, 2003.

[13] Douglas Thain and Miron Livny. Error scope on a computational grid: Theory and practice. In *Proc. of the 11th IEEE HPDC*, page 199, Washington, DC, USA, 2002. IEEE Computer Society.

[14] G. Tsouloupas and M. D. Dikaiakos. GridBench: A workbench for grid benchmarking. In P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *EGC*, volume 3470 of *LNCS*, pages 211–225. Springer, 2005.

[15] Rob V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency & Computation: Practice & Experience.*, 17(7-8):1079–1107, June-July 2005.

Deployment and Interoperability of Legacy Code Services

Y. Zetuny[1], G. Kecskemeti[1], T. Kiss[1], G. Sipos[2], P. Kacsuk[2], G. Terstyanszky[1], S. Winter[1]

[1]*Centre of Parallel Computing,Cavendish School of Computer Science,*
University of Westminster, 115 New Cavendish Street, London W1W 6UW,
[2]*MTA SZTAKI Laboratory of Parallel and Distributed Systems*
H-1518 Budapest, P.O. Box 63, Hungary

Abstract. The Grid Execution Management for Legacy Code Architecture (GEMLCA) enables exposing legacy applications as Grid services without re-engineering the code, or even requiring access to the source files. The integration of current GT3 and GT4 based GEMLCA implementations with the P-GRADE Grid portal allows the creation, execution and visualisation of complex Grid workflows composed of legacy and non-legacy components. However, the deployment of legacy codes and mapping their execution to Grid resources is currently done manually. This paper outlines how GEMLCA can be extended with automatic service deployment based on Grid brokering, and information system. A conceptual architecture for an Automatic Deployment Service (ADS) and for an Interoperable Bridge Service (IBS) are introduced explaining how these mechanisms will improve new releases of GEMLCA.

1. Legacy Code Services for the Grid

The Grid requires Grid-enabled applications capable of utilising the underlying middleware and infrastructure. Most of the current Grid applications are either significantly re-engineered applications or new ones. However, as the demand for Grid computing is increasing in both business and scientific communities, there is a need for porting a vast legacy of applications to Grid computing. Companies and organisations cannot afford to throw legacy code applications away for the sake of a new technology, and there is a clear business imperative to migrate the existing applications onto the Grid with the least possible effort and cost. Grid computing is moving to a point where reliable Grid middleware and high-level tools will be offered to support the creation of production Grids. A high-level Grid toolkit should definitely include components for converting legacy applications into Grid services.

The Grid Execution Management for Legacy Code Architecture (GEMLCA) [1] enables legacy code programs written in any source language (Fortran, C, Java, etc.) to be easily deployed as a Grid Service 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 applications onto the Grid.

In order to offer a user-friendly application environment, and support the creation of complex Grid applications, GEMLCA is integrated with the workflow oriented P-GRADE Grid portal [2]. Using the integrated GEMLCA – P-GRADE portal solution users can create complex Grid workflows from legacy and non-legacy components, map them to the available Grid resources, execute the workflows, and visualise and monitor their execution.

A drawback of the current GEMLCA solution is the static mapping of legacy components onto resources. Before creating the workflow the legacy application has to be deployed, and during workflow creation, but prior to its submission, the user has to specify the resource where the component will be executed. It would be desirable to allocate resources dynamically at run-time and to automatically deploy a legacy component on a different site in order to achieve better performance. The automatic service deployment raises interoperability issues in policy and security management, which should be also addressed.

Figure 1 illustrates how GEMLCA can be extended with these functionalities. Instead of mapping the execution of workflow components statically to the different Grid sites, the abstract workflow graph created by the user is passed to a resource broker together with Quality of Service (QoS) requirements. The broker contacts an information service and tries to map different components of the workflow to different resources and pre-deployed services. If users' QoS requirements cannot be fulfilled with the currently deployed services, or if the required service is not deployed on any of the resources, the broker contacts the automatic deployment service in order to deploy the code on a different site. As the sites can belong to different Grids with different middleware, policy and security standards, the deployment service should resolve these interoperability problems.

Unfortunately no currently existing information system, broker or deployment service can be directly used and integrated with GEMLCA to solve these problems. Significant research, extension and improvement of existing solutions are necessary. In this paper we concentrate on a subset of this complex architecture and propose a solution for the Automatic Deployment Service (ADS) and for an Interoperable Bridge Service (IBS).

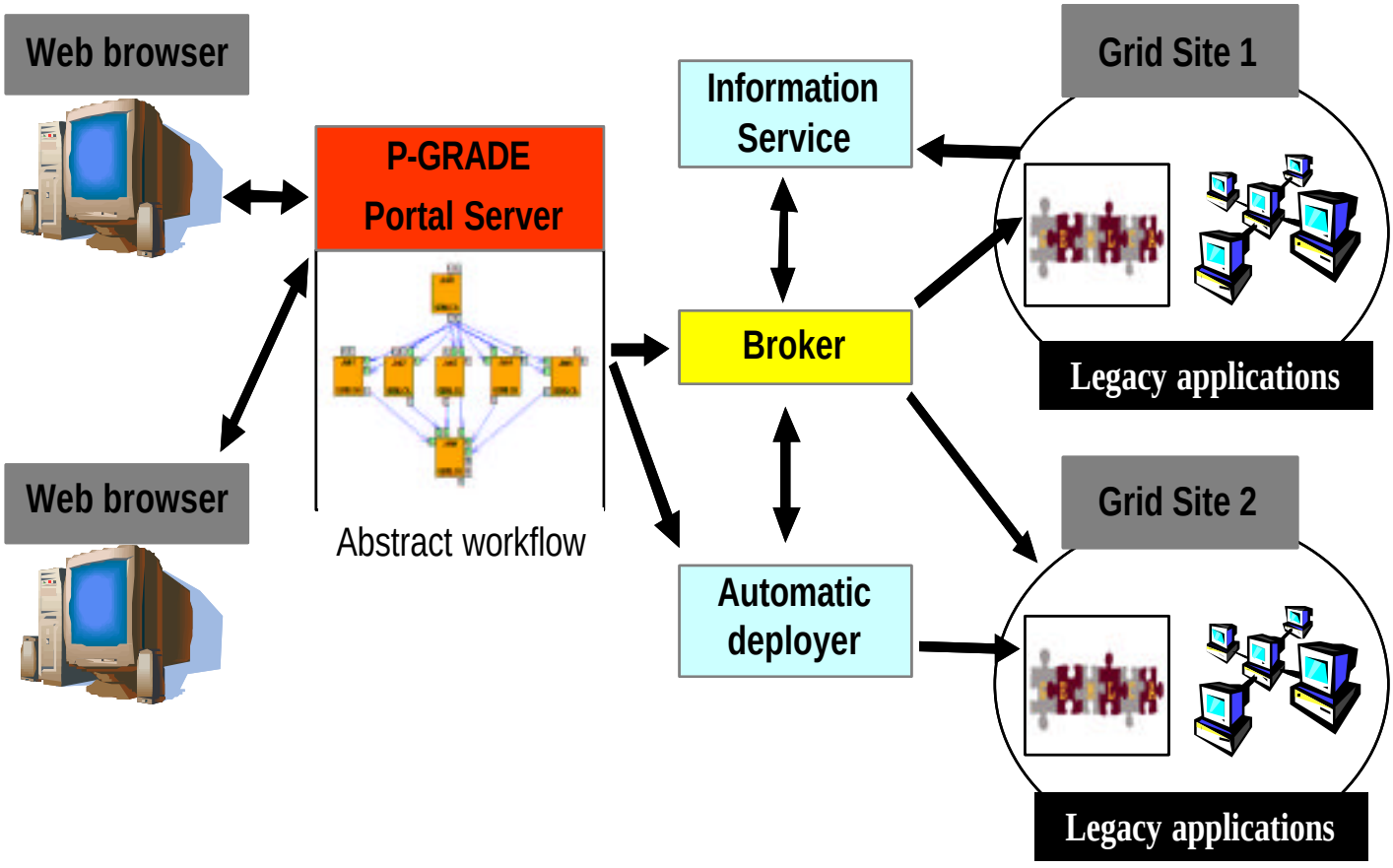


Figure 1: Brokering, information system and automatic deployment support in GEMLCA

2. Automatic Deployment Service in GEMLCA

In the current GEMLCA architecture legacy code services are deployed and mapped manually to Grid resources at workflow construction time. As a pre-requisite to extending GEMLCA with QoS-based brokering and load-balancing capabilities, services have to be automatically deployed or migrated from one site to another. This section describes the challenges faced when deploying services, and proposes a general architecture for an Automatic Deployment Service (ADS).

2.1 Deployment Scenarios

There are several research efforts identifying and implementing solutions for scenarios where automatic deployment of services is important [3]. Each scenario can be derived from the following two basic cases:

1. *Deploying new Grid services*. In this scenario new Grid services are deployed onto new sites. Dependencies have to be detected and resolved by the automatic service deployment tool, and the service container has to be prepared in order to prevent misbehaviour.
2. *Migrating existing Grid services*. This scenario occurs when deployed Grid services are transferred to different sites using their dependency descriptions. However, even within the same Grid, this description could be in a different format than required, depending on the selected service container. An automatic deployment tool should provide a transformation between different dependency descriptions. Where the description is not appropriate, dependencies have to be investigated like in the previous scenario.

Based on these two basic scenarios the following use cases demonstrate where to use automatic service deployment in a Grid environment:

- *Automatic selection services.* If a deployed service cannot process any more request as its hosting container is overloaded, the service has to be migrated to a site with lower workload, and some of the service requests have to be redirected to the newly deployed service.
- *Grid integration.* Grids could be more efficient as a result of lower communication overhead when those services, which need a lot of communication, are installed inside the same Grid.
- *Refining existing services.* Some services, for example data retrieval solutions, provide very generic information for users, which could be irrelevant. In this case users have to filter this information in order to retrieve what is relevant for them. To avoid high network traffic the filtering can be implemented and deployed as a new service on the site where the Grid service is installed.

2.2 Deploying Services

Service deployment incorporates the following steps: collecting site and service descriptions, classifying sites based on these descriptions, checking dependencies of the classified sites, comparing service and site descriptions, selecting one of the sites where to

deploy the service, finally deploying the service. Further, we describe how to deploy a service after selecting the site.

After selecting a site the service deployer has to detect the available sandboxing techniques on the site. Each site should provide a set of minimal security requirements for the sandbox. This set contains the enforcement constraints of the site [6], which should define the maximal usage of the available hardware resources, for example: available CPU cycles, network bandwidth, memory, etc. These constraints help to avoid the improper usage of the resources like DOS attacks initiated from the site.

For example in a Linux environment the service deployer automatically checks the kernel extensions [13] for accounting and fine-grained process separation (e.g. gr security patches). Next, the ADS checks the available sandboxing solutions taking into account the constraints for the site and the service itself. There are two options:

a) Each execution environment management has its own interface to enforce and monitor the actual environment. The service deployer has to make sure that the sandbox preparation tasks are not interfering with the future usage of the service with overusing the resources available for the service. The sandbox preparation includes the installation of the required service container in the virtual environment.
b) If the site follows an indulgent policy and the service under deployment is a Java code then the service deployer may decide not to use the available sandboxing solutions. This can speedup the deployment process dramatically since there is no need to prepare a whole service container and its dependencies, just the service itself. In this case the service deployer prepares a special classloader for each service which will restrict the access of other classes on the system and enforce the security policies of the site with the installation of a security manager.

After selecting one of the sandboxing solutions and the service deployer creates the sandbox using one of the existing execution environment management services. Currently, the following service-oriented execution environment managers are available: GT3-based RTEFactory [4] and Dynamic Virtual Environment services [5], GT4-based Workspace Management Service and WS-based VMPlants [7]. Finally, the service deployer transfers the service from the source site to the destination site.

2.3 Deployment Service Architecture

In order to support the automatic service deployment a layered deployment service architecture has been developed. Figure 2 shows this architecture, and illustrates how ADS migrates a deployed service to a new site. The migration process and the tasks of the different layers of the architecture are the following:

1. The Grid sites register themselves in an information system. The registration contains basic site descriptions.
2. In order to be migrated from *site A* to another site, the service contacts the ADS.
3. The deployment service queries the information system in order to access site descriptions, and also generates the description of the service to be migrated. Based on these descriptions, which are transformed into a meta-description [9], the classifier [8] checks the description of the service against the site descriptions, and generates a set of sites that are capable of hosting the service. Next, the dependency checker investigates the capabilities of the selected sites. The capabilities should be

identified with a black box method as the source code is not available in GEMLCA. In this method, dependencies are detected using an observer execution environment. The service uses generic test data that affects all of its features in order to gather runtime dependencies, such as the files accessed, network connections used, or environment variables needed to be set up. The generated descriptions are stored in the information system for further use.

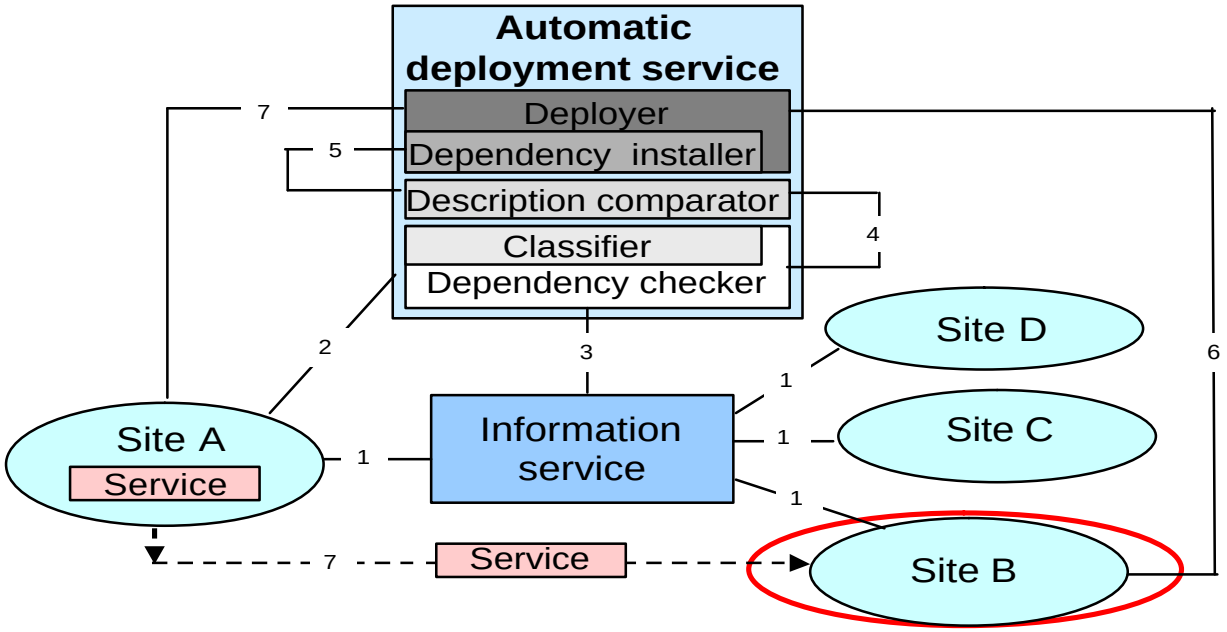


Figure 2: Automatic Deployment Service Architecture

4. Using information received from the dependency checker the description comparator analyses some metrics, such as cost and time requirements of the deployment based on the descriptions, and selects the site with the lowest deployment cost, for example *Site B* [10].
5. In order to make *Site B* compatible with *Site A* from the service's point of view, the dependency installer prepares several installation scripts and environment configuration files/setup scripts. These scripts have to take care of all third party software necessary for the service. The established network connections have to be simulated with a proxy. This proxy has to be prepared on both sites.
6. The deployer prepares a sandbox [11] on *Site B* in order to separate the execution of the service from others. The sandboxing technique used can be various; e.g. a basic *chroot*ed environment, some Java security model based solution, or a virtualisation technique (Xen, VirtualPC, VMware). The deployer creates a new sandbox, and then the installation scripts are executed in it.
7. The deployer notifies *Site A* and negotiates the transfer of the service between the *Site A* and *Site B*. The deployer can detect the available and accessible transfer services on each site. It also has the capability to act as an intermediate layer between the source and the destination sites, if it is necessary. The service has to be registered with the new host environment in an execution environment specific way without restarting it. [12]).

After the transfer is completed between the two sites the service becomes available on the new site.

3. x-Service Interoperability

The ADS presented in the previous section offers solution for automatic deployment of services within the same administrative domain. However, interoperability issues have to be taken into consideration when bridging different Grid domains. The aim of our Grid services interoperability research is to build on existing policy and security solutions and standards that are managed independently by different Grid sites, and to develop an architecture that is capable of bridging in a flexible, scalable and dynamic manner. As a result of this work, GEMLCA will be significantly extended to enable the deployment, creation, invocation and management of Grid services between multi-domain Grid environments to support a dynamic integration of different Grid sites.

3.1 Authorisation and Interoperability

Authorisation is essential in Grid applications since it defines the process of deciding whether an entity can access a particular resource or service. In contrast to authentication, it is not feasible to manage authorisation on a local site basis. The reason for that is the fact that users have direct administrative agreements only with their own local sites and with the Virtual Organisation (VO) they work in, but not with other entities. Therefore, to be able to access different sites, users' authorisation data should be kept and managed in a central repository eliminating the risk of data corruption and the burden of managing multiple identical authorisation data in different sites on the Grid. In practice, from an authorisation point of view, a Grid is established by enforcing agreements between the resource providers (RP) and the VO where resource access is controlled by both parties with different roles. This situation can lead into two problems in the authorisation management. The first problem is that there is no clear role separation between what authorisation information should be kept at the VO level and what at the RP level. The second problem, impacted by the first problem, is that there is a high risk that dual roles could be created at both levels. Analysing these two problems we made a conclusion that the authorisation data should be divided into two categories: VO and RP information. The VO information attempts to answer the question "What is a user allowed to do at a VO?" whereas the RP information attempts to answer the question "What is a user allowed to do at an RP". In order to manage these two categories, VO information should be contained in a server managed by the VO itself while the RP information should be contained at the RP local site near the resources and controlled by some kind of an extended ACL.

In order to address these problems, major research efforts have created several authorisation services integrated with VO. In general, there are two kinds of authorisation services which are currently available: attribute authorities (AA) and policy assertion (PA) services. The VO Membership Service (VOMS) [14], which was developed by the European DataGrid Project (EDG), is an example of an attribute authority service. Users have X.509 proxy certificates that contain the user's information from the VOMS servers with additional authorisation information, such as role and group information. Since VOMS does not include a built in policy engine, attribute certificate (AC) information must be extracted by a relying party (resource) and evaluated against its local policy. In contrast to VOMS, Permis [15] and the Globus Community Access Service (CAS) [16] are examples of policy assertion services, both of which can issue statements encoded in SAML. Although Permis contains a powerful policy engine, AC information is kept in a single AC repository in contrast to VOMS where AC information is distributed to the user. This

feature limits the flexibility of Permis which may lead to problems in a VO-oriented environment. In contrast, CAS does not issue ACs, but whole new proxy certificates with the authorisation information included in an extension. As a consequence, when a service receives a certificate, it can not determine who the owner is without inspecting the extension. As a result, Globus-based Grids would need to be modified to use a CAS certificate. An additional drawback in CAS is the fact that it does not record groups or roles, but only permissions. Akenti [16] is another example of a policy assertion based authorisation service. However, it does not use true ACs, since their definition and description do not conform to the standard. In addition to that, Akenti is targeted on authorising accesses on web resources (such as websites) which makes it less appealing to be used in a VO-oriented environment.

The implementation of the described authorisation services leads into two conclusions. The first conclusion is that none of the authorisation services currently available can provide a complete answer to the authorisation problem and secondly, an ideal authorisation solution should be able to accommodate various access controls models and implementations.

3.2 Interoperability Bridge Service (IBS)

General interoperability architecture, the IBS, has been specified in order to handle interoperability issues between Grid clients and Grid services when they are in different domains ("x" refers to any kind of Grid or Web service in this context). One of the requirements for the IBS is to be able to support different authorisation services which may be used at the different Grid sites. Also, it may require that the different services providing access to resources would be able to describe their authorisation policies in a standard manner so that the bridge would have clear semantics of what is required to access a particular service.

Extending the ADS with IBS enables the automatic deployment of a Grid service into different domains. IBS serves as a bridge between different Grids, and makes the deployment to a different domain transparent for the ADS by redirecting the communication between the ADS and the services through IBS, as illustrated on Figure 3. The architecture is composed of five layers:

Negotiator Layer - collects interoperability properties, such as access mechanisms, policies, and security mechanisms of the involved domains.

Analyzer Layer - analyses the properties collected by the negotiator layer, defines the differences between domains, and prepares a list of interoperability requirements based on these differences.

Classifier Layer - classifies the interoperability requirements into interoperability classes. It utilizes a mapping engine to create correlation between the demands of each domain.

Dispatcher Layer - uses the mapping produced by the classifier layer to spawn a Bridge Service that contains the generated mappings. Each dispatched bridge includes a unique identifier which is then can be used by a client to access the service.

Bridge Layer - encompasses one or more Bridge Services that are spawned by the Dispatcher Layer. Each Bridge Service is intended to resolve a particular interoperability problem. The Bridge service is discarded once a communication is no longer required.

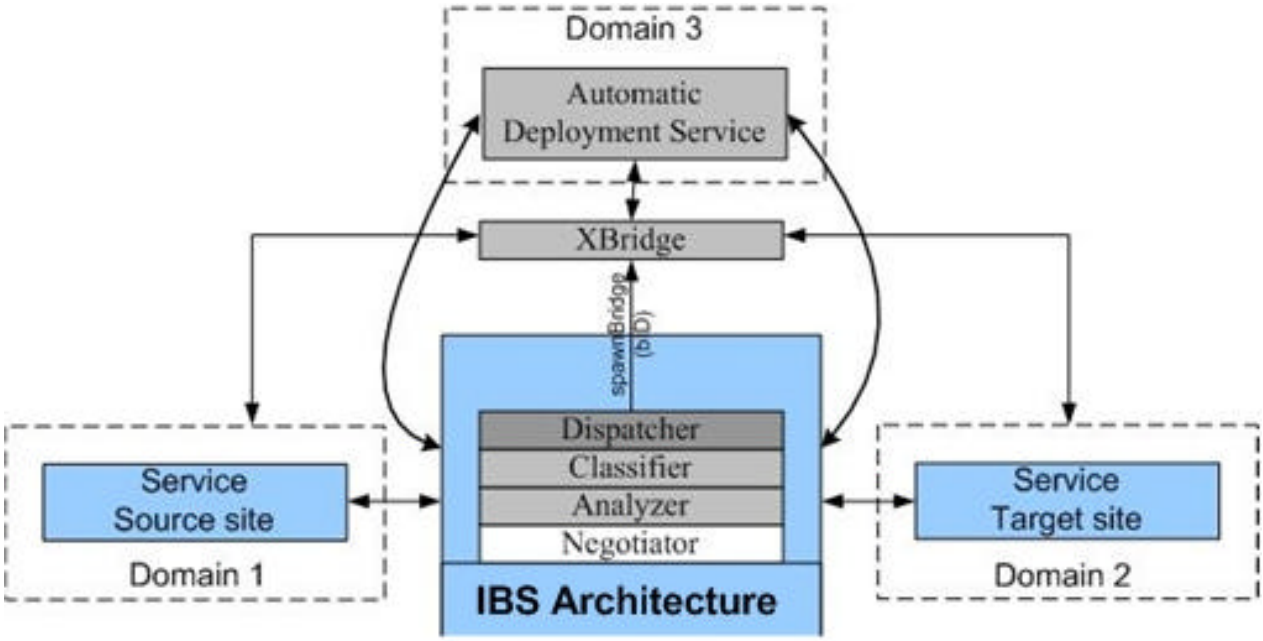


Figure 3 ADS and the x-Service Interoperability

The following figure presents a scenario where the IBS is contacted by the ADS to deploy a service and the IBS has to manage the authorisation aspect between two Grid sites belonging to different Grid domains.

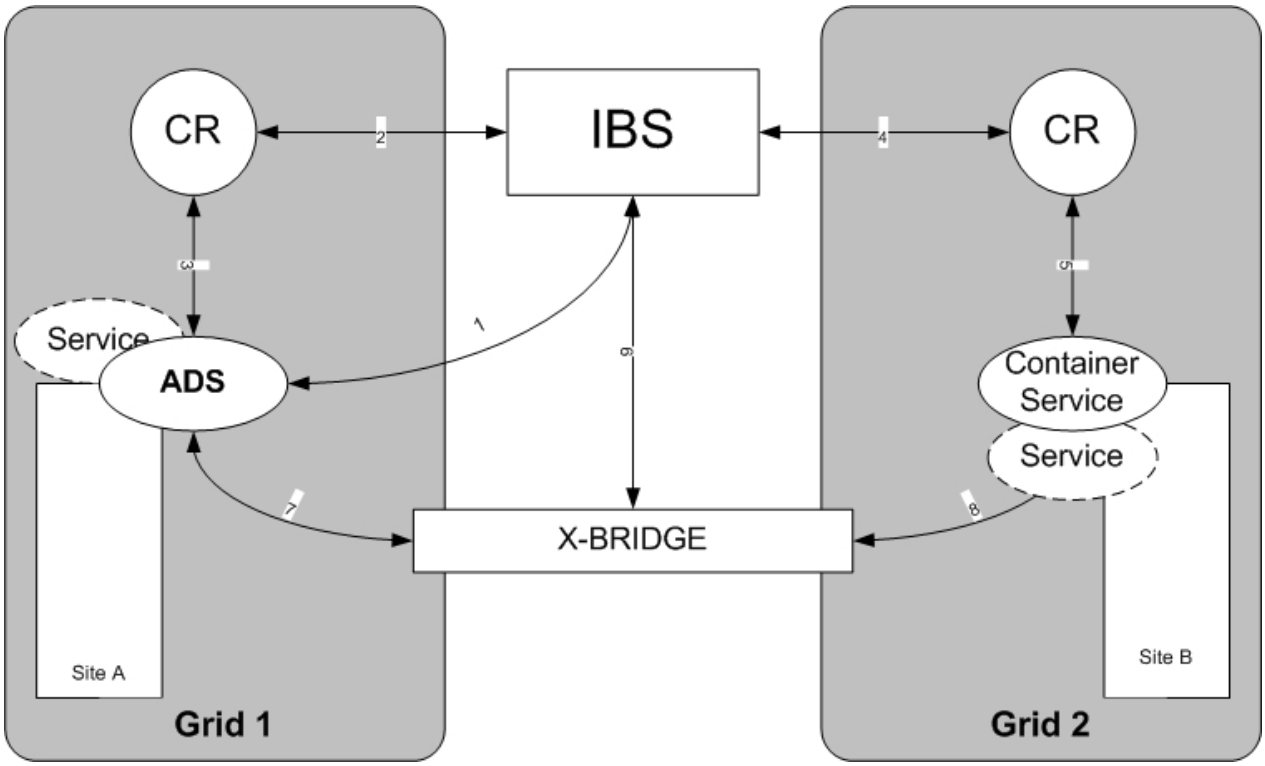


Figure 4: Interoperability Scenario

In step 1, the ADS contacts the IBS requesting to deploy a *Service1* onto a *Site B* in Grid 2. The IBS contacts a central security repository (CR), such as VOMS (step 2), to retrieve authorisation policies of the ADS (step 3). It is possible that the CR will not have a built in

policy engine and in that case it will have to query a policy engine first. The same steps are repeated for the Container Service in order to retrieve its authorisation policy (steps 4 and 5). It is important to note that both the client and the service can have different authorisation policies and both of them have to be taken into account. Based on the policy descriptions, the IBS analyses, maps the differences and generates the X-BRIDGE service (step 6), which is able to translate the process between the ADS and the Container Service. The X-BRIDGE service acts as both service and client since it provides a remote interface for the ADS itself. It contacts the container service on behalf of the ADS. In order to achieve this communication, the IBS has to notify the ADS about the availability of the X-BRIDGE service. In step 7 and step 8 the ADS contacts the X-BRIDGE in order to deploy the service on *Site B*. Once the deployment process is completed, the IBS is informed and it removes the bridge.

4. Conclusion and Further Work

Deploying legacy applications on the Grid without reengineering the code is crucial for the wider scientific and industrial take-up of Grid technology. GEMLCA provides a general solution in order to convert legacy applications as black-boxes into OGSA compatible Grid services, without any significant user effort. Current GEMLCA implementations fulfil this objective, and the integrated GEMLCA - P-GRADE Portal solution offers a user friendly Web interface and workflow support on top of this. However, GEMLCA should be further developed and extended with additional features, like information system support, brokering, load balancing or automatic deployment and migration of services, in order to offer a more comprehensive solution for Grid users.

This paper presented an Automatic Deployment Service (ADS) architecture that enables the automatic deployment and migration of GEMLCA Grid services to different sites within the same Grid domain. The combination of this architecture with the Interoperable Bridge Service (IBS) extends deployment and migration capabilities to different Grid domains. Adding these features to GEMLCA enables service developers to deploy their services automatically on the target site, or to migrate the service to a different site, spanning multiple Grid domains when required, if execution is more efficient there. The implementation of these architectures and their integration with GEMLCA is currently work in progress. Also, the investigation has already started how it could be integrated and extended with existing information system and brokering solutions in order to realise the full GEMLCA-based Grid presented on Figure 1.

References

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

[2] Cs. Nemeth, G. Dozsa, R. Lovas, P. Kacsuk, "The P-GRADE Grid portal", In: Computational Science and Its Applications - ICCSA 2004: International Conference, Assisi, Italy, 2004, LNCS 3044, pp. 10-19.

[3] J. B. Weissman, S Kim, D. England. Supporting the Dynamic Grid Service Lifecycle, Technical Report, University of Minnesota, 2004,

[4] Kate Keahey, Matei Ripeanu and Karl Doering. Dynamic Creation and Management of Runbtime Environments in the Grid. Workshop on Designing and Building Web Services (GGF9), October 2003.

[5] Katarzyna Keahey, Karl Doering and Ian Foster. From Sandbox to Playground: Dynamic Virtual Environments in the Grid. 5th International Workshop on Grid Computing (Grid 2004). November 2004.

[6] Fangzhe Chang, Ayal Itzkovitz and Vijay Karamcheti. User-level Resource-constrained Sandboxing. USENIX Windows Systems Symposium. August 2000.

[7] Ivan Krsul, Arijit Ganguly, Jian Zhang. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. 2004

[8] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. Second Symposium on Operating Systems Design and Implementation. October 1996.

[9] Mike James: Classification Algorithms, Wiley, 1985, ISBN: 0-471-84799-2

[10] M. Cannataro, C. Comito: A Data Mining Ontology for Grid Programming, Conf. Proc of the 1st Workshop on Semantics in Peer-to-Peer and Grid Computing at the Twelfth International World Wide Web Conference, 20 May 2003, Budapest, Hungary

[11] P. Watson, C. Fowler. An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet, Technical Report, University of Newcastle, February, 2005.

[12] M. Smith, T. Friese, B. Freisleben. Towards a service-oriented ad hoc grid, Conf. Proc of the ISPDC/HeteroPar Conference, 2004

[13] A. Ting, W. Caixia, X. Yong. Dynamic Grid Service Deployment, Technical Report, March, 2004

[14] R. Alfieri, R Cecchini, V. Ciaschini, L dell' Agnello, A Frohner, A. Gianoli, K. Lorentey, F. Spataro: VOMS: an Authorisation Ssystem for Virtual Organisations

[15] R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello., A. Gianoli, F. Spataro: Managing Dynamic User Communities in a Grid of Autonomous Resources, Computing in High Energy and Nuclear Physics, 24-28 March 2003, La Jolla, California

[16] L. Pearlman, V. Welch, I. Foster, K. Kesselman, S Tuecke: A Community Authorization Service for Group Collaboration, IEEE Workshop on Policies for Distributed Systems and Networks, 2002

[17] http://www-itg.lbl.gov/Akenti

Correctness of a Rollback-Recovery Protocol for Wide Area Pipelined Data Flow Computations

Jim Smith, Paul Watson

University of Newcastle upon Tyne

Abstract. It is argued that there is a significant class of pipelined large grain data flow computations whose wide area distribution and long running nature suggest a need for fault-tolerance, but for which existing approaches appear either costly or incomplete. An example, which motivated this paper, is the execution of queries over distributed databases. This paper presents an approach which exploits some limited input from the application layer in order to implement a low overhead recovery protocol for such data flow computations. Over a large range of possible data flow graphs, the protocol is shown to support tolerance of a single machine failure, per execution of the data flow computation, and in many cases to provide a greater degree of fault-tolerance.

1 Introduction

The suitability of data flow for computations which process a succession of inputs in pipeline fashion has long been appreciated [9]. Early work sought to exploit very fine grain parallelism in special data flow architectures. However, this entails high bandwidth interconnect, so later work aimed to increase the grain size, trading off some degree of parallelism for an easier realisation. This trend manifested itself both in automated processing of special purpose data flow languages and in manual parallelization of essentially sequential code. While any larger grain approach is likely to suit a more loosely coupled architecture, such as networks of autonomous machines, manual approaches appear to be most used. There are a number of infrastructures which support gluing together of pure functions, e.g. [6], and dynamic scheduling of the resulting *digraphs*. Within applications however, the support for stateful *vertices* as offered in systems such as [13] is often assumed, for instance to aggregate several token values, or to meaningfully combine tokens from several streams.

The use of large grain data flow techniques has become established in database query processing [12]. In the context of distributed databases, [5] makes a case for an open form of distributed query processing where participants contribute not just data sources but also functionality and cycle providers. As described in [15], the emergence of computational grids provides much support and motivation for the evolution of this kind of open query processing. In this open environment, many widely distributed and autonomous resources may be combined into the execution of any particular query.

Much emphasis has been given recently to query processing over continuous streams [2]. Typical stream processing systems may access widely distributed data, and may employ parallelism, but are essentially centralized. By contrast [19] anticipates query plans being distributed over multiple sites, perhaps linking together stream resources made available by separate organisations.

Publish subscribe systems are long running and manipulate streams of events, distributing them over a wide area to a potentially very large number of subscribers. Both the dissemination and the filtering may be distributed over a number of sites [4] and there is suggestion that testing for the correlation of multiple events may be desirable [3]. One strategy to meet the requirements for supporting asynchrony would be to maintain the necessary storage of events at the leaves of the tree, which would be geographically closer to the subscribers. The non-leaf nodes would then form a simple wide area data flow graph.

Such pipelined dataflow applications have requirements for wide area distribution and stateful *vertices*, yet also have requirements for fault-tolerance. This work shows that existing rollback-recovery techniques are inapplicable, incomplete, or likely to prove expensive when applied to such computations. To address this gap, a large class of *digraphs*, which seem likely to be the most commonly used, is identified. These *digraphs* have properties that inspire a family of protocols which can exploit limited input from the application level to provide low overhead fault-tolerance support. The work goes on to define and verify a detailed proposal for the simplest of these protocols.

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 defines a model for a distributed large grain data flow computation. Section 4 presents the rollback-recovery protocol in the context of this model and establishes the correctness of its behaviour and Section 5 concludes.

2 Related Work

Of protocols surveyed in [10], the most suitable for wide area appears to be the log-based protocols which avoid the need to coordinate checkpoints of individual processes by logging indeterminate events, i.e. messages. However, they all rely on checkpointing process state, all-be-it independently, to support pruning of the recovery logs. Such checkpoints must be made to a location where they can be accessed by whichever machine takes over the work of a failed machine. In a wide area context, this could be a single machine located far from many of the processes, or multiple separate, but local, machines. The protocol described here is a log-based protocol, but exploits some additional input from the user level to obviate the need for checkpointing of process state.

Replication based support for fault-tolerance in software based data flow systems is described in e.g. [8], but only for stateless *vertices*. It is possible to support replication of stateful *vertices* through multicasting of messages and replicated processing [14] or by copying state between coordinator and cohorts. The overhead in either case is likely to be high particularly in a wide area. The protocol described here supports recovery for stateful *vertices*, yet avoids both

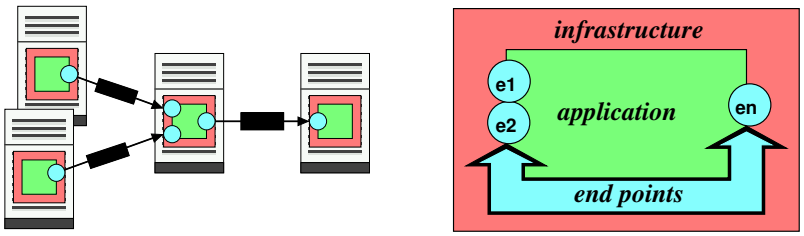
repeated transmission of tokens and copying of *vertex* state in normal running in order to reduce overhead, at the cost of more expensive recovery.

In systems which assume pure functional *vertices* to permit dynamic scheduling of activations to processors, it is possible to preserve tokens used by an activation remote from the executing processor until the activation has safely written its result tokens. This allows for retry if the executing processor fails. However, emulating stateful *vertices* in such systems is likely to be expensive. A development of this theme is to retain multiple tokens in an upstream *vertex* thereby allowing replay of arbitrary amounts of the computation. Marker tokens, e.g. *flow tuples* [7] can be inserted into the output stream to coordinate the arrival of tokens downstream with their purging from logs upstream. This potentially allows restoration of *vertex* state, but such earlier work has not defined a protocol for purging logs. In the absence of such a protocol, it is always necessary for a recovering machine to replay the whole execution prior to the failure. This paper presents a complete log-based protocol and establishes its correctness properties.

3 Computational Model

3.1 Data Flow

Figure 1 shows mapping of an example data flow graph onto machines in a network and the partitioning of software within a *vertex*. A *vertex* contains



(a) An example graph. (b) A single vertex.

Fig. 1. Computational model for large grain data flow.

some arbitrary ***application*** processing which can transform, generate or delete tokens, based on those it receives via certain ***end points***. The application can direct result tokens to subsequent *vertices* via other *end points*. The *end points* are represented as an array of objects whose operations call on the services of an underlying ***infrastructure*** layer. The model adheres to the common notion of large grain data flow in [13]. *Vertices* are statically scheduled to machines, thereby permitting the application code in a *vertex* to exploit local memory, e.g. to aggregate token values.

The application code of a *vertex* is shown in Figure 2. Each loop iteration retrieves a single token, performs local processing and outputs all consequent

```
app() {
    do {
        Token t = get_next_token(); // call receive() on end point(s)
        process(t);                 // do any local processing
        output_results();           // call send() on some end point(s)
    } while (! finished_processing());
}
```

Fig. 2. Main loop in application.

result tokens. As is typical in large grain data flow, the application code in a *vertex* is not triggered by a globally defined firing rule. The choice as to which *edge* to receive a token from is assumed to be defined within the application specific operation *get_next_token()*; if required the underlying call on the infrastructure will block. It is then assumed that the order in which the application code within a *vertex* processes is deterministic, so that the application code in a *vertex* will perform consistently, independent of the order in which tokens arrive on incoming *edges*.

The interface exported by an *end point* can be characterised by the following operations.

send(input Token) is called by the application code to transmit a token which is destined for the *end point* at the opposite end of the edge.

dosend(input Token) is a helper function which encapsulates a call on the infrastructure layer to transfer a token to the other end of the *edge*. Typically, when this call returns the token is not yet at the other end of the *edge* but is buffered in the infrastructure for subsequent transfer.

receive(): returns Token is called by the application to retrieve the next available token from the *end point*'s input queue.

handle(input Token) is called by the infrastructure on arrival of a token destined for this particular *end point*, to deposit the token into the *end point*'s input queue.

3.2 Fault-Tolerance

The following assumptions are made.

- Loss or corruption of individual messages is masked in the infrastructure service, e.g. through use of a reliable transport such as TCP.
- Machine failures, e.g. due to power failure or reboot, are detected within the infrastructure layer. An example implementation of an infrastructure level fault-detection service for use in a the wide-area is described in [18].
- A replacement machine can be found, and integrated into the system by the underlying infrastructure, e.g. from a pool of spare machines or through

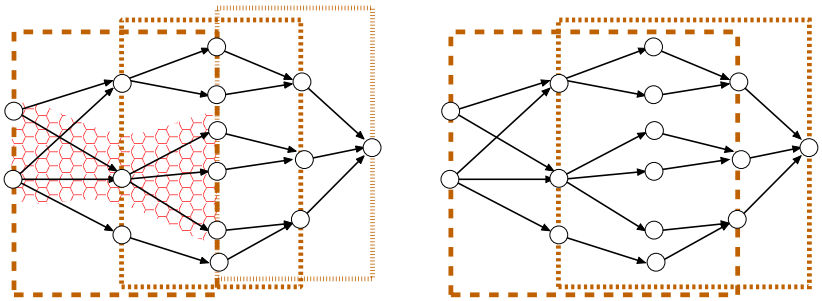
dynamic acquisition. Integration of a standby to replace a failed machine can be seen as ensuring that for each surviving machine which will need to communicate with the new one, the mapping between logical participant and physical machine identification is updated. Such integration is described in the context of the well known message passing infrastructure MPI in [11].

- Machines are not required to have stable storage; buffering employed by the rollback-recovery protocol can take place in volatile memory which is initialised at (re)start. Where space restrictions necessitate spooling recovery log data onto local disk, such spooled data is also discarded in restart.
- This work does not present support for tolerance to failure of *vertices* which interact with the real world. One possibility is for a *source* to log all input received onto stable storage before processing it and for a *sink* to save each result onto stable storage before outputting it.

4 The Rollback-Recovery Protocol

4.1 Uniform Graphs

In [16], a digraph is referred to as *uniform* if all paths between any two vertices are of the same length. Example *uniform* graphs are shown in Figure 3. Such graphs can be partitioned into overlapping *slices* as shown. *Slices* may be further partitioned into *segments* where the boundary of a *segment* passes through a subset of the *vertices* on each of upstream and downstream boundaries of a *slice* and where no *edge* crosses a *segment* boundary. Where the *slice thickness* is 2, *segments* have a particularly simple form, as shown by the example in the figure.



(a) *slice thickness* 2 (b) *slice thickness* 3

Fig. 3. Example *uniform graph*, showing *slices* of different *thickness* and in (a) a example segment.

The protocol described here relies on the return of acknowledgments from downstream *vertices* in order to support the truncation of a recovery log which contains tokens sent out along a particular *edge*.

4.2 Checkpoint Marker Tokens

Acknowledgment of data tokens is accomplished by *end points* inserting checkpoint marker tokens into their output streams. Each checkpoint marker carries a sequence number which is unique for its creator. An *end point* increments its sequence number with each checkpoint marker creation. Tokens lying between a pair of checkpoint markers can be thought of as a *block* marked by the checkpoint marker immediately following. To acknowledge receipt of all tokens up to a given checkpoint marker, it is only necessary to return the checkpoint marker itself. Figure 4 shows a possible structure for such a checkpoint marker. The shaded

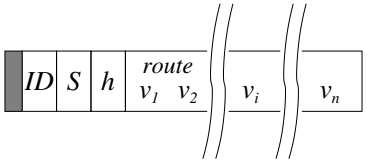


Fig. 4. A checkpoint marker token.

box represents a field which distinguishes this token as a checkpoint marker. *ID* is a value which may be used to distinguish its creating *end point*. It is employed by *end points* of downstream vertices to identify which saved checkpoint marker a newly received one should be compared with; an *end point* need only keep the latest checkpoint marker it receives from any upstream *end point*. *S* is the sequence number. *h* indicates the number of hops (i.e. *vertices*), the checkpoint marker should travel downstream before it is acknowledged. The entries *v1... vn* define the route taken by the checkpoint marker in its forward path. The list is traversed in the opposite direction as the checkpoint marker retraces that path to its originator as an acknowledgment. Within a *slice*, checkpoint markers are generated and acknowledged in *sources* and *sinks* of that *slice* respectively.

4.3 Rollback-Recovery using *Slices* of *thickness* 2

The *end point send()* operation is modified to copy all tokens and checkpoint markers to the recovery log. It is also modified to increment a local counter *tcount* (initially zero) when processing each counter. When the local counter *tcount* reaches a set value, a new checkpoint marker is generated using the instance variable *sequence*, which is incremented, and *tcount* reset to zero.

Figure 5 shows how checkpoint and acknowledgment tokens can be processed on arrival at an *end point*. If this *end point* is the tail of an out-going *edge*, the token received may be of two types.

acknowledgment If the acknowledgment is for a checkpoint generated here, all data and checkpoint tokens up to the corresponding checkpoint marker can be purged from *recoverylog*. Otherwise the sequence number in it is recorded

```
handle(Token t) {
    if (t.kind == acknowledgement) {
        if (−−t.hopcount < 0) recoverylog.purgeTo(t.sequence);
        else newest_ack[t.sender] = t; // app code collates
    } else if (t.kind == restart) {
        t.kind = flushed; dosend(t); replay_recovery_log()
    } else if (t.kind == flushed) {
        flushing = false; discard_any_entries_in_bufferq();
    } else if (!flushing) { // flag is true at startup
        if (t.kind == checkpoint) {
            if (−−t.hopcount <= 0) {
                t.hopcount = t.path.entries(); // i.e. distance travelled
                t.kind = acknowledgement;
                dosend(t);
                if (newer(t.sequence, remembered_sequence[t.source])) {
                    move_entries_in_bufferq_to_inputq();
                } else discard_corresponding_entries_in_bufferq();
            } else {t.path.push(myid); bufferq.enqueue(t);}
        } else bufferq.enqueue(t);
    } // else discard token
}
```

Fig. 5. Processing checkpoint marker and acknowledgment in *handle()*.

for use by the application layer which defers relay of any acknowledgment until a copy has been received from each downstream adjacent *vertex*.

restart request Following a *flushed* token, the contents of the recovery log, including checkpoint markers, are replayed.

If this *end point* is the head of an in-coming edge, then the arriving token may be one of the other three types.

flushed This signals start of the recovery proper.

application data The token is enqueued in a holding buffer.

checkpoint marker If the checkpoint marker is destined for a downstream *vertex*, the *ID* of this *end point* is added to the path stored in the checkpoint marker before the latter is enqueued in the holding buffer. Otherwise the entries in the holding buffer are moved to the input queue and the checkpoint marker sent as an acknowledgment back upstream.

The *end point receive()* operation need not change to support recovery. It is seen above that arriving tokens are not copied into *inputq* until a corresponding checkpoint marker arrives. Thus, *receive()* only returns when a whole block has been moved into *inputq*.

Figure 6 shows modifications made to the application code to exploit the fault-tolerance provision of the rollback-recovery protocol. At startup, a restart request is sent to each upstream adjacent *vertex*. The essential processing of data

```
app() {
    for (u in upstream_vertices) endpoint[u]->send(restart);
    do {
        Token t = get_next_token(); // call receive() on end point(s)
        if (! t.checkpoint) {
            process(t);              // do any local processing
            output_results();        // call send() on some end point(s)
        } else newest_cp[t.sender] = t;
        for (u in upstream_vertices) {
            if (done with tokens up to newest_cp[u])
                {for (e in outgoing_endpoints) e.send(newest_cp[u]);}
            Token ack = null; // compares as oldest
            for (v in downstream_vertices)
                ack = oldest(endpoint[v].newest_ack[u], ack);
            if (newer(ack, last_ack_sent[u]))
                {endpoint[u]->send(ack); last[u] = ack;}
        }
    } while (! finished_processing());
}
```

Fig. 6. Enhancing the application code to handle acknowledgments.

tokens remains unchanged. However, the application code must coordinate the forwarding of checkpoint tokens and relay of acknowledgments. This processing is clearly application specific. Figure 6 shows a rather general case where the latest checkpoint number for each upstream adjacent *vertex* and *acknowledgment* for each downstream adjacent *vertex* are maintained in two arrays and both checkpoint marker and acknowledgment processing is performed in the loop over upstream adjacent *vertices* at the bottom of the main application loop. The code only relays an acknowledgment to an upstream adjacent *vertex* when all downstream adjacent *vertices* have acknowledged it.

The correctness of the protocol when operated within a single *segment* of a *slice* of *thickness* 2 in a *uniform* graph is examined below. In the context of this discussion, ***sources*** and ***sinks*** are those of the *segment*, and not necessarily those of the containing graph.

1. A token is only released for processing by the application layer of any ***sink*** when the checkpoint marker first following it has been received there, and an acknowledgment sent. FIFO ordering of tokens within an *edge*, and the correct processing of checkpoint markers by application code, ensure that the checkpoint marker inserted by a ***source*** into a token stream to mark a *block* of tokens will arrive directly after (the subset of) those tokens at the ***sinks***.
2. All tokens are preserved in the ***sources*** until acknowledged by all ***sinks***.
3. Following from 1 and 2, if the *central vertex* fails, the ***sources*** are guaranteed to hold, in their recovery logs, all tokens which had been sent to the failed

vertex but not acknowledged, or passed to the application layer, in the ***sinks***. There may be tokens in these logs which had been received by all ***sinks***, but for which the acknowledgment had not yet reached the ***sources***.

4. If the *central vertex* restarts, or fails and is replaced, then the application layer in that *vertex* requests restart from all ***sinks***.
5. Each incoming *end point* in the *central vertex* discards all tokens received after startup until receipt of the first token of type *flushed*; such tokens will have been in transit at failure.
6. In recovery, the new or restarted *vertex* reprocesses all tokens in any block not acknowledged by all ***sinks***.
7. Provided the processing which takes place in the application layer is deterministic, the new or replacement *vertex* will generate the same output tokens, as were generated before failure, for any work which it redoes during recovery; the ***sinks*** will be able to recognise a repeated block.
8. The ***sources*** and ***sinks*** can support failure of the *central vertex* even while recovery is in progress.
9. Since the *segments* of a *slice* of *thickness* 2 are independent the computation in a slice of *thickness* 2 is 1-fault-tolerant with regard to the *inner vertices* of that *slice*.

If a *digraph* is covered by overlapped *slices* of *thickness* 2, clearly non-adjacent *slices* do not overlap each other and the bounding of the protocol ensures that concurrent failure of any single *inner vertex* within each of a pair of non-overlapping *slices* can be tolerated. Taking account of overlapping *slices* [16] the protocol only guarantees to tolerate a single failure during the overall computation; the restriction from 1-fault-tolerance reflecting a trade-off of fault-tolerance for lower protocol overhead which is characteristic of the protocol.

5 Conclusions

It has been argued in this work that there is a class of applications which naturally suit a pipelined large grain data flow expression, but which through being long running and distributed over autonomous resources in a wide area, require provision for fault-tolerance. General purpose approaches to fault-tolerance seem likely to incur a high cost, particularly in a wide area context, e.g. through checkpointing potentially large state to remote sites.

This work has presented a protocol suited for a range of *digraphs*, specifically *uniform graphs* in which there is no pair of paths which have different lengths between any pair of *vertices*. Such graphs can be partitioned into overlapping *slices* of some defined *thickness*, the maximum distance between a *source* and *sink* of a *slice*. Tokens are acknowledged a fixed distance from the *vertex* where they are generated. The protocol is then bounded within a *slice* of the graph, such that tokens are generated in its *sources* and acknowledged in its *sinks*. The correspondence between recovery log position and acknowledgment is established by the insertion of *checkpoint markers* into the token stream; these are returned as acknowledgments. The operation of a protocol is described for a *slice* of *thickness*

2 and shown to support 1-fault tolerance for the *inner vertices* of a *slice* and, by overlapping such *slices*, to tolerate at least a single fault in an arbitrary *uniform digraph*. Ongoing work is investigating: protocols to support a *thicker slice* and cost models to a quantitative comparison with alternative fault-tolerance strategies. The protocol has been implemented and shown to have low overhead [17] in an enhancement to the distributed query processing system OGSA-DQP [1] developed in collaboration with Manchester University.

References

1. N. Alpdemir, A. Mukherjee, A. Gounaris, N. W. Paton, P. Watson, and Alvaro A. A. Fernandes. OGSA-DQP: A grid service for distributed querying on the grid. In *EDBT*, 1979.
2. S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, September 2001.
3. J. bacon, K. Moody, and J. Bates et al. Generic support for distributed applications. *Computer*, June 2000.
4. G. Banavar, T. Chandra, and B. Mukherjee et al. An efficient multicast protocol for content-based publish subscribe systems. In *ICDCS*, 1999.
5. R. Braumandl, M. Keidl, and A. Kemper et al. Objectglobe: Ubiquitous query processing. *VLDB Journal*, August 2001.
6. J. C. Browne, J. Werth, and T. Lee. Experimental evaluation of a reusability-oriented parallel programming environment. *IEEE TSE*, February 1990.
7. M. Cherniack, H. Balakrishnan, and M. Balazinska et al. Scalable distribute stream processing. In *CIDR*, 2003.
8. R. Davoli, L-A Gianchini, and Ö. Babaoglu et al. Parallel computing in networks of workstations with paralex. *IEEE TPDS*, April 1996.
9. J. B. Dennis and k. S. Weng. An abstract implementation for concurrent computation with streams. In *ICPP*, 1979.
10. E. N. Elnozahy, L. Alvisi, and Y-M Wang et al. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 2002.
11. G. Fagg and J. J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI User's Group Meeting*, 2000.
12. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, June 1993.
13. R. Babb II. parallel processing with large grain data flow techniques. *Computer*, July 1984.
14. M. Shah, J. M. Hellerstein, and S. Chandrasekaran et al. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.
15. J. Smith, A. Gounaris, and P. Watson et al. Distributed query processing on the grid. In *International Workshop on Grid Computing*, November 2002.
16. J. Smith and P. Watson. A rollback-recovery protocol for wide area pipelined data flow computations. Technical Report CS-TR-836, University of Newcastle, 2004.
17. J. Smith and P. Watson. Fault-tolerance in distributed query processing. In *IDEAS*, 2005.
18. P. Stelling, C. DeMatteis, and I. T. Foster et al. A fault detection service for wide area distributed computations. *Cluster Computing*, 1999.
19. S. Zdonik, M. Stonebraker, and M. Cherniack et al. The aurora and medusa projects. *DE Bulletin*, March 2003.

Towards Integration of Legacy Code Deployment Approaches*

B. Balis[4], M. Bubak[4], A. Harrison[3], P.Kacsuk[2], T.Kiss[1], G. Sipos[2], I. Taylor[3]

[1] Cavendish School of Computer Science, University of Westminster, London
[2] MTA SZTAKI Lab. of Parallel and Distributed Systems, Hungary
[3] Cardiff School of Computer Science, Cardiff University, Cardiff
[4]Inst. of Computer Science and Academic Computer Centre CYFRONET AGH, Krakow
Corresponding Author T. Kiss - T.Kiss@westminster.ac.uk

Abstract. While current trends in grid computing are moving towards workflow-based service-oriented architectures, most of the computations are still done by legacy codes. In this paper we describe different approaches to deployment of legacy code in a grid environment. First, we present several tools that enable the deployment and user access of legacy codes at three different levels: user views (portals), legacy deployment tools/frameworks, and execution environments (e.g., monitoring tools). Next, we present three realistic scenarios in which execution of legacy code on the grid is involved. Finally, we conclude with an integrated scenario, which consolidates legacy-application tools to address highly dynamic environments.

1 Introduction

As the Grid computing paradigm gains in momentum and the supporting middleware becomes more stable, so the expectations of users become more sophisticated. Moving forward from simple task-farming parameter sweep applications in which codes are farmed across a set of distributed resources, users are now looking at new, more complex scenarios involving interaction between service-based environments offering diverse capabilities. Much of the actual computation on grids is done by legacy codes – software written before grids became a possibility. Enabling the use of these codes within this context is a major challenge for middleware developers. In some situations a user may wish to view a legacy code as a black box which simply receives and returns files. Under other conditions the user may wish to control execution in a more fine-grained manner, making specific calls to legacy libraries, for example. While these different approaches to exposing legacy code have their own challenges, a key feature of existing and emerging scenarios is the need to compose codes into workflows. This requires that the different techniques used to expose legacy codes can be linked together, providing a seamless flow of control and data. Furthermore, once a workflow has been constructed, the user may wish to monitor progress of the composition at runtime. This monitoring may result in reconfiguration of the workflow or spawning new workflows.

This paper describes existing tools for handling legacy codes. Each has its own user-groups and scenarios. Our motivation is to explore how the integration of these

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

tools can substantially extend their individual capabilities and hence the scenarios they support. We describe four scenarios – the first three showing current capabilities, while the final scenario, an extension of existing capabilities, demonstrates the need for further integration.

2 Existing Tools

This Section describes the existing tooling developed by participating parties. They are a mixture of applications that deal with: (1) User Views, (2) Legacy Code Deployment, (3) Execution Environment. We look at each in turn.

2.1 User Views

Grid portals and workflow engines provide a high level abstraction of the legacy application that is specifically tailored to the needs of the end user. Below we describe two.

2.1.1 Triana

Triana [7] is a workflow-based graphical problem solving environment initially developed for gravitational-wave data analysis but has been extended to a variety of domains: it now includes over 500 Java tools for signal, image and audio processing and statistical analysis. Additionally, Triana can interact with service-oriented Triana components; that is, applications that can be invoked via a network interface, such as P2P, WSRF and Web services. It is also fully integrated with the GAT [11], providing the ability to interact with Grid-oriented Triana components to execute applications on the Grid via a Grid resource manager (such as GRAM, GRMS or Condor/G) and perform operations that support these applications, such as file transfer (e.g. GridFTP).

Distributed components obviously increase massively the power of workflows that can be created within Triana as large compute resources remote to the user can be employed, as can third-party applications. However, local Java tools still play a vital part in Triana, even when distributed components handle the majority of a computation. Any combination of distributed and local components can be connected within a single workflow to create complex data-driven Grid workflow scenarios, connecting distributed services, Grid jobs and scripts for intelligent decision-making abilities making Triana applicable to dynamic scenarios. Triana also contains support for non-intrusive legacy-code wrapping, through its integration with gridMonSteer [9].

2.1.2 P-GRADE Portal

The P-GRADE portal [1] is a workflow-oriented Grid portal with the main goal to cover the whole lifecycle of workflow-oriented computational grid applications. It enables the graphical development of workflows consisting of various types of executable components (sequential, MPI or PVM programs), executing these workflows in

Globus-based grids relying on user credentials, and finally analyzing the correctness and performance of applications by the built-in visualization facilities.

Workflow applications can be developed in the P-GRADE portal by its graphical Workflow Editor.

A P-GRADE portal workflow is an acyclic dependency graph that connects sequential and parallel programs into an interoperating set of jobs. The nodes of such a graph are jobs, while the arc connections define the execution order of the jobs and the data dependencies between them that must be resolved by the workflow manager during the execution.

Managing the transfer of files and recognition of the availability of the necessary files is the task of the workflow manager portal subsystem, currently implemented on the top of Condor DAGMan [3]. The workflow manager is capable to transfer data among Globus VOs, thus the different components of the same workflow can be mapped onto different Globus VOs. These VOs can be part of the same grid, or can belong to multiple grids.

2.2 Legacy Code Deployment

This Section describes two tools for the exposure of legacy codes as services that can be combined into higher level workflows.

2.2.1 GEMLCA

GEMLCA [2] 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 high-level GEMLCA conceptual architecture is composed of four basic components:

1) The **Compute Server** is a single or multiple processor computing system on which several legacy codes are already implemented and available. The goal of GEMLCA is to turn these legacy codes into Grid services that can be accessed by Grid users.
2) The **Grid Host Environment** implements a service-oriented OGSA-based Grid layer, such as GT3 or GT4. This layer is a pre-requisite for connecting the Compute Server into an OGSA-built Grid.
3) The **GEMLCA Resource** layer provides a set of Grid services which expose legacy codes as Grid services.
4) The fourth component is the **GEMLCA Client** that can be installed on any client machine through which a user would like to access the GEMLCA resources.

The deployment of a GEMLCA legacy code service assumes that the legacy application runs in its native environment on a Compute Server. It is the task of the GEMLCA Resource layer to present the legacy application as a Grid service to the user, to communicate with the Grid client and to hide the legacy nature of the application. The deployment process of a GEMLCA 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 deployment defines the execution environment and the parameter set for the legacy application 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 [2].

2.2.2 LGF

Legacy to Grid adaptation Framework – LGF [8] is a framework that aids developer in deploying legacy code as web or grid services. LGF provides a set of tools for automatic generation of grid-service codes based on the description of legacy code interface prepared by the developer. The generated code consists of a couple of grid (web) services which can be deployed in a usual manner after which the legacy code is available through a WS-Interface. Most of the code is generated automatically, though the developer has to implement the actual invocations of legacy code manually. Though in this way the process is not fully automatic, it allows for maximum flexibility as to the way the legacy code is used. The invocations may be realized as simply as calls to library functions or as complex as communication to a remote system.

In LGF, the WS-Interface is decoupled from the legacy code. There are three elements to the system.

1. The Container, where the grid services exposing WS-Interface to the legacy code are deployed.
2. A pool of Backend Systems, where legacy code is executed.
3. The Clients which request the legacy code services through the Container.

An instance of the legacy code, deployed in the Backend System, registers in the Container. Subsequently, when a Client invokes a WS method, one of the legacy code instances is assigned to handle the request. In this way multiple legacy code instances can serve one WS front-end. This in turn enables several interesting features such as dynamic deployment of legacy code, resource brokering, dynamic load balancing or fault tolerance. For example, processing may be transparently switched from one backend system to another when a load on the original backend system exceeds a threshold. Last but not least, in this architecture a new instance legacy code can be dynamically deployed on a new Backend System via a simple job submission.

2.3 Execution Environment

This Section describes different tools that support execution related tasks of the legacy service, like monitoring, steering or automatic migration and deployment of applications.

2.3.1 gridMonSteer

gridMonSteer [9] provides a simple, non-intrusive way to integrate legacy applications as components within Triana workflows. As the name suggests, it provides two

fundamental roles: monitoring of output files (at the legacy application level) and steering (at the application and/or workflow level). gridMonSteer monitors the application while it is running; intermediate results can be passed back to the flow, making it ideal for constructing complex workflows, where dynamic interactions are commonplace. The gridMonSteer architecture was born from a collaboration between Cardiff University and Center of Computation and Technology (CCT), LSU, which investigated the integration of distributed Cactus [6] simulations within Triana workflows. From this research, a Grid-friendly Cactus thorn was developed to provide the distributed connectivity from Cactus to a Triana component, allowing intermediate results to be fed from Cactus into a running workflow. This interaction was demonstrated during the Supercomputing 2004 conference and gridMonSteer is a generalisation of this architecture.

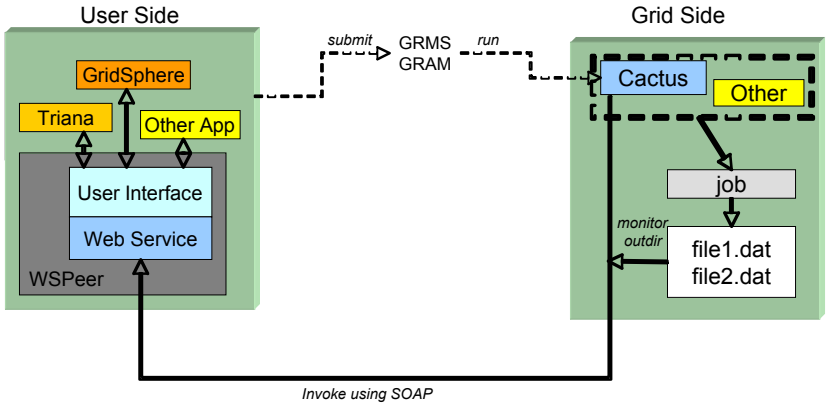


Fig. 4. gridMonSteer Architecture [9]

As illustrated in Figure 4, gridMonSteer consists of a Grid side *application wrapper*, which executes a legacy application, monitors specified directories for files created by the application and then notifies the application controller accordingly; and a user side application controller, which can be any application that wishes to receive input from application wrappers. The controller, in our case, could be Triana, GEMCLA or P-Grade. Triana has been used previously to demonstrate these capabilities by integrating the dynamic deployment capabilities of WSPeer [5] to expose a Web service interface that implements the gridMonSteer protocol for notification and delivery of the distributed files.

2.3.2 Mercury Monitor

The architecture of Mercury Monitor is based on the Grid Monitoring Architecture (GMA) proposed by Global Grid Forum (GGF), and implemented in a modular way with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system. The input of the monitoring system consists of measurements generated by sensors.

Sensors are controlled by producers that can transfer measurements to consumers when requested. Sensors are implemented as shared objects that are dynamically loaded into the producer at run-time depending on configuration and incoming requests for different measurements.

In Mercury all measurable quantities are represented as metrics. Metrics are defined by a unique name (such as, host.cpu.user which identifies the metric definition), a list of formal parameters and a data type. By providing actual values for the formal parameters a metric instance can be created representing an entity to be monitored. A measurement corresponding to a metric instance is called metric value.

Metric values contain a time-stamp and the measured data according to the data type of the metric definition. Sensor modules implement the measurement of one or more metrics. Mercury Monitor supports both event-like metrics (i.e. an external event is needed to produce a metric value) and continuous metrics (i.e. a measurement is possible whenever a consumer requests it such as, the CPU temperature in a host). Continuous metrics can be made event-like by requesting automatic periodic measurements. In addition to the functionality proposed in the GMA document, Mercury also supports actuators.

Actuators are analogous to sensors but instead of taking measurements of metrics they implement controls that represent interactions with either the monitored entities or the monitoring system itself. Besides monitoring this also facilitates steering.

2.3.3 ADS: Automatic Deployment Service

Service deployment makes services available on resources where they were not available previously. Automatic service deployment is especially challenging in service-oriented Grid middleware because resources and services have to be added, modified and removed dynamically. The Automatic Deployment Service (ADS) Architecture [4] has been defined including the following layers:

- *inspector layer*, to detect dependencies of the service to be deployed,
- *classifier layer*, to create an ontology-based classification of site and service descriptions,
- *comparator layer*, to investigate a destination site and estimate the deployment costs,
- *installer layer*, to create a sandbox for the service code to be deployed,
- *deployer layer*, to make the service available in a service-oriented Grid.

As ADS is limited to the deployment of a Grid service in a single domain Grid environment, a general interoperability architecture, the x-Service Interoperability Layer (XSILA), has also been specified in order to handle interoperability issues between Grid clients and Grid services when they are in different domains. Extending the ADS with XSILA enables the automatic deployment of a Grid service into different domains. XSILA serves as a bridge between the different Grids, and makes the deployment to a different domain transparent for the ADS by redirecting the communication between the ADS and the services though XSILA.

3 Application Scenarios

We will now present four scenarios which reveal how users can have differing requirements depending on the nature of the scenario and the legacy codes employed in it. The first scenario deals with coarse-grained (applications), the second with fine-

grained (libraries) legacy codes. The third scenario illustrates how legacy systems such as databases can also be deployed as a grid resource. Finally, a scenario is presented where the combination of these different approaches is required.

3.1. Traffic Simulation – Adaptation of Legacy Applications

A traffic simulation application is typically built from different functional building blocks. For example one module generates a road network file that describes the topology of a road network and the allowed junction manoeuvres on the roads. A second component, the actual simulator, simulates car movements in time using the previous network file as input. The results of the simulation, typically a trace file, can be loaded into different visualiser and analyser tools. In a realistic scenario traffic analysts wish to run several simulations to analyse the effects of changing network parameters, like traffic light patterns, one way streets or the effects of increased traffic on particular roads. They create a workflow where the results of the road network generator are fed into several simulator components, and then the outputs are sent to analyser/visualiser components. This requires parameter study like execution of several simulations and their subsequent analysis. Distribution appears at two different levels. The components of the workflow could be either sequential or parallel applications that require a computer cluster. Also, some components of the workflow can be executed parallel to each other on different Grid resources.

These legacy components of the workflow have to be deployed on several Grid sites and exposed as Grid services. The source codes of these legacy applications are typically not available (especially if we are talking about commercial products) resulting in a need for coarse grained black-box type wrapping like the one provided by GEMLCA. Once the components are expressed as Grid services workflow can be created using a workflow engine. The different components of the workflow can be mapped either statically at workflow creation time, or dynamically at run-time to the available resources. The user interface for workflow creation, execution and visualisation is a Grid portal, like the P-GRADE portal.

3.2 Dynamic Data Driven Application Scenario (DDDAS)

The complete astrophysics DDDAS example is taken from an advanced scenario developed at Cardiff and the Center for Computation and Technology (CCT) through previous work [9] and involves the following steps:

1. Scientific Workflow: Triana is used to specifying the staging of the distributed scenario, graphically within a workflow. This includes specifying where each Cactus simulation is submitted for distributed execution, to specify the input/output channels for such simulations and for analysis of the results, at various stages during the scenario.
2. Startup: the Cactus application is launched on the distributed resource. This could be accomplished in a number of ways using the GAT [10] e.g. the GAT could decide to use GRAM, condor or more rudimentary tools, such as SSH. Along with the Cactus simulation, Triana will instantiate a unit that will dy-

namically launch a Web service to receive file notification and the incoming data from the application, using the protocol described previously.
3. Simulation Begins: the simulation is launched.
4. Cactus Output: Cactus discovers the address (endpoint) of the client, which wishes to receive its output and tries to connect. This is achieved through specified Unicast addresses at present but could be extended to utilise some kind of caching e.g. UDDI, or similar.
5. Detection of Events : the demonstrator detects something interesting happens in the simulation e.g. an apparent horizon of a coalescing black hole binary system.
6. ApplicationSteering: the demonstrator then makes a decision based on the stimuli or evolution of the application and dynamically instantiates a workflow to aid in the further investigation of this aspect.
7. JobSpawning: the resulting workflows then perform multiple parallel Job submissions of Cacti across the Grid of available resources to perform a distributed search across the parameter range.
8. Results: as the Cacti finish their individual runs, they return the results to the main steering application, which returns the optimal results to Triana for final visualization or for steering the main Cactus simulation.

DDDAS scenarios, such as these, are generally Grid unaware, they would require advanced portal interfaces or workflow system, with intelligent resource brokers, to locate the resources and deploy the legacy codes on-the-fly as and when they are needed. Since the workflows are highly data dependent, they should be able to be constructed on-the-fly depending on the type of analysis that is required and farmed out accordingly depending on resources. Further, since real-time data analysis would be required, the application would need to monitor legacy codes when they are running and would therefore need capabilities similar to gridMonSteer, described in section 2.3.1.

3.3 Adapting Legacy Systems into Grid-enabled Workflows

In some scenarios, it may become necessary to access external systems, such as a database, an external analysis system, or a monitoring system, to fetch input, store output or to analyse the collected information. For example, this is the case in the two scenarios described above. In Traffic Simulation, the final output of analyses, or even intermediate outputs of simulation, could be stored in a repository to allow for a later processing, e.g. results of the analysis of a multiple series of parameter studies simulating traffic with different parameters could be saved and later retrieved for a comparative analysis. The Gravitational Wave Analysis scenario explicitly requires that the final outputs are stored in a database for subsequent analysis.

To enable flexible construction of grid workflows in such cases, legacy systems need to be exposed as grid services. This is achieved using the LGF – Legacy-to-Grid Adaptation Framework. LGF allows that with minimum assistance from the developer, a legacy system, such as database, can be deployed as a web/grid service and accessed through a WS-interface. In case of databases, it is important that the WS-interface for the database supports transactional interaction with a database. LGF

supports transactional processing by exposing a standard set of WS-methods for transactional interactions if this is requested. Once a WSDL description for this interface is available and the service is deployed, the integration with a portal (in the sense of composing the legacy system into a workflow) will be straightforward.

3.4 Integration of Existing Tools

The final scenario introduces some challenges that need to be addressed. The current tooling described cannot handle this kind of scenario in an efficient way and, to the user, a seamless manner. In particular this scenario raises certain issues:

1. In scenario 2 the running of Cactus is presumed – what if there are no nodes on the grid that have Cactus installed?
2. If the previous question proves false, where are we to retrieve Cactus from?
3. And once we have retrieved it, how do we install and execute it?
4. Once installed, how do we monitor its output files during execution?

These issues above relate to the dynamic spawning of workflow as much as they do to the initialisation of, and dynamic interaction with, the processes. We believe that through the integration of the systems and tools described above, this scenario can be realised. Below we describe the process.

1. The user view component searches for a node that has Cactus installed but was unsuccessful.
2. The user view component instructs GEMLCA to deploy the required code and specifies the required runtime systems.
3. GEMLCA locates a suitable node and deploys the code (using ADS) by using the parameters supplied by the user view component (i.e. which VO etc)
4. ADS asks LGF, which acts as a code repository and therefore has access to the packaged, deployable code bundles for the required code -- this includes runtime environment capabilities defined by the user view component.
5. ADS deploys the code bundle retrieved from LGF and notifies GEMLCA.
6. GEMLCA exposes the deployed code as a service and notifies the user view component of successful deployment of both the executable code and runtime environment requirements.
7. The user view component invokes the newly deployed service and makes use of its chosen execution environment.
8. Once running, the gridMonSteer wrapper can monitor output files and distribute its progress to the user.

4 Conclusion and Future Work

As grid computing aims to facilitate more and more complex, workflow-based applications, the inclusion of legacy code components in these workflows remain a key issue. There are several approaches to present legacy components as services and include them in grid workflows. However, these solutions are concentrating on specific user requirements and cover only a subset of possible scenarios. On the other hand, some of these approaches complement each other very well and the combina-

tion of these in a component framework would provide significant enhancements of current capabilities.

In this paper we described several tools at three different levels that facilitate legacy code deployment and their inclusion in workflows. The first three scenarios described current capabilities and were the result of separate research efforts. However, as it was proven by scenario 4, further integration of these solutions could significantly enhance their capabilities and extend the scenarios supported. The aim of this paper was to explore the available tools and their current capabilities, and to define how further integration could improve these. Work is currently in progress of specifying the common framework how this integration can actually be implemented and will be published in a separate paper.

References

[1] G. Sipos and P. Kacsuk: Classification and Implementations of Workflow-Oriented Grid Portals, To appear in the Proc. of The 2005 International Conference on High Performance Computing and Communications (HPCC2005), Sorrento, Italy

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

[3] T. Tannenbaum, D. Wright, K. Miller, and M. Livny: Condor - A Distributed Job Scheduler. Beowulf Cluster Computing with Linux, The MIT Press, MA, USA, 2002.

[4] G. Kecskemeti, Y. Zetuny, G. Terstyanszky, S. Winter, T. Kiss, P. Kacsuk: Automatic Deployment and Interoperability of Grid Services, To appear in the Conf. Proc. of the UK E-Science All Hands Meeting, 19 - 22 September 2005, Nottingham, UK

[5] The WSPeer framework, http://www.wspeer.org/index.html

[6] Cactus computational toolkit, http://www.cactuscode.org.

[7] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. Journal of Grid Computing, 1(2):199–217, 2003.

[8] B. Balis, M. Bubak, M. Wegiel. A Solution for Adaptating Legacy Code as Web Services. In Proc. Workshop on Component Models and Systems for Grid Applications. 18th Annual ACM International Conference on Supercomputing, Saint-Malo, France, July 2004.

[9] T. Goodale, I. Taylor and I. Wang. Integrating cactus simulations within Triana workflows. In Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies, pages 47–53, 2005.

[10] G. Allen, K. Davis, K.N. Dolkas, N.D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor. Enabling applications on the Grid: A GridLab overview. International Journal of High Performance Computing Applications, 2003. Special issue on Grid Computing: Infrastructure and Applications.

[11] Gabrielle Allen et. al. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid, Submitted to IEEE, http://www.gridlab.org/WorkPackages/wp-1/Documents/Allen2.pdf

User Friendly Legacy Code Support for Different Grid Environments and Middleware [1]

T.Kiss[1], G. Sipos[2], G.Terstyanszky[1], T.Delaitre[1], P.Kacsuk[2], N. Podhorszki[2], S.C. Winter[1]

[1] Centre for Parallel Computing, Cavendish School of Computer Science
University of Westminster,115 New Cavendish Street, London, W1W 6UW
[2] MTA SZTAKI Lab. of Parallel and Distributed Systems,
H-1518 Budapest, P.O. Box 63, Hungary

Abstract. The more widespread academic and industrial take-up of Grid technology requires user friendly Grid application environments where users can utilise their existing legacy code programs as Grid services, create new Grid-enabled applications and submit jobs or complex workflows to available Grid resources. This paper describes how the integration of two different tools, the GEMLCA legacy code support solution and the P-GRADE workflow oriented Grid portal, fulfils most of these objectives. The integrated GEMLCA P-GRADE portal environment provides legacy code publication and execution support for a large scale of Grid architectures and middleware.

1 Introduction

There are many efforts all over the world to provide new Grid middleware concepts for constructing large production Grids. As a result, the Grid community is in the phase of producing third generation Grid systems that are represented by the OGSA (Open Grid Services Architecture) and WSRF (Web Services Resource Framework) specifications. On the other hand relatively little attention has been paid to how end-users can survive in the rapidly changing world of Grid generations. Moreover, the efforts in this field remained isolated resulting only in limited functionality prototypes for specific user domains and not serving a wider user community.

This paper describes how the integration of two different architectures, the P-GRADE Grid portal [1] and GEMLCA (Grid Execution Management for Legacy Code Architecture) [2] resulted in a more generic solution serving a large variety of Grid systems and application domains. The integrated solution provides a high-level user-friendly Grid application environment that supports users of GT2-based second generation and GT3/GT4-based third generation Grid systems from the same user interface. It also allows the integration of legacy code applications into complex Grid

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

workflows which can be mapped to Grid nodes running this wide variety of middleware.

The integration has happened at different levels. In the first step, the GEMLCA clients were added to the portal providing a user-friendly interface for legacy code deployment, execution and visualisation. On the other hand this integration also enhanced the usability of the originally GT2-based P-GRADE portal making it capable to handle GT3/GT4 Grid services. In the second step, GEMLCA was extended to handle legacy code submission to current GT2-based production Grids, like the UK National Grid Service or the EGEE Grid. This resulted in a legacy code repository that makes it even easier to P-GRADE portal end users to create and execute workflows from previously published legacy components.

Current joint activities are concentrating on two different areas:

- porting the integrated solution to desktop Grid systems and extending the functionality of these Grids towards service support,
- creating a more loosely coupled integration that allows incorporating advanced functionality, like GEMLCA-based legacy code support, into the portal as a plug-in, resulting in more flexible solution depending on actual user requirements.

The paper introduces GEMLCA and the P-GRADE portal and describes the integration activities outlined above.

2 Baseline technologies

2.1 P-GRADE Portal

The P-GRADE portal [1] is a workflow-oriented Grid portal with the main goal to cover the whole lifecycle of workflow-oriented computational grid applications. It enables the graphical development of workflows consisting of various types of executable components (sequential, MPI or PVM programs), executing these workflows in Globus-based grids [4] relying on user credentials, and finally analyzing the correctness and performance of applications by the built-in visualization facilities.

Workflow applications can be developed in the P-GRADE portal by its graphical Workflow Editor.

A P-GRADE portal workflow is an acyclic dependency graph that connects sequential and parallel programs into an interoperating set of jobs. The nodes of such a graph are jobs, while the arc connections define the execution order of the jobs and the data dependencies between them that must be resolved by the workflow manager during the execution. An example for P-GRADE portal workflows can be seen in the middle part of Figure 2. Large rectangles represent jobs while small rectangles around the jobs are called ports and represent data files that the corresponding jobs expect or produce. Directed arcs interconnect pairs of input and output files if an output file of a job serves as an input file for another job.

The semantics of the workflow execution means that a job (a node 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 user-

defined input files are available either on the portal server and at the pre-defined grid storage resources. Therefore, the workflow describes both the control-flow and the data-flow of the application.

Managing the transfer of files and recognition of the availability of the necessary files is the task of the workflow manager portal subsystem, currently implemented on the top of Condor DAGMan [3]. The workflow manager is capable to transfer data among Globus VOs [4], thus the different components of the same workflow can be mapped onto different Globus VOs. These VOs can be part of the same grid, or can belong to multiple grids.

2.2 GEMLCA

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 high-level GEMLCA conceptual architecture is represented on Figure 1.

As shown in the figure, there are four basic components in the architecture:

1) The **Compute Server** is a single or multiple processor computing system on which several legacy codes are already implemented and available. The goal of GEMLCA is to turn these legacy codes into Grid services that can be accessed by Grid users.
2) The **Grid Host Environment** implements a service-oriented OGSA-based Grid layer, such as GT3 or GT4. This layer is a pre-requisite for connecting the Compute Server into an OGSA-built Grid.
3) The **GEMLCA Resource** layer provides a set of Grid services which expose legacy codes as Grid services.
4) The fourth component is the **GEMLCA Client** that can be installed on any client machine through which a user would like to access the GEMLCA resources.

The deployment of a GEMLCA legacy code service assumes that the legacy application runs in its native environment on a Compute Server. It is the task of the GEMLCA Resource layer to present the legacy application as a Grid service to the user, to communicate with the Grid client and to hide the legacy nature of the applica-

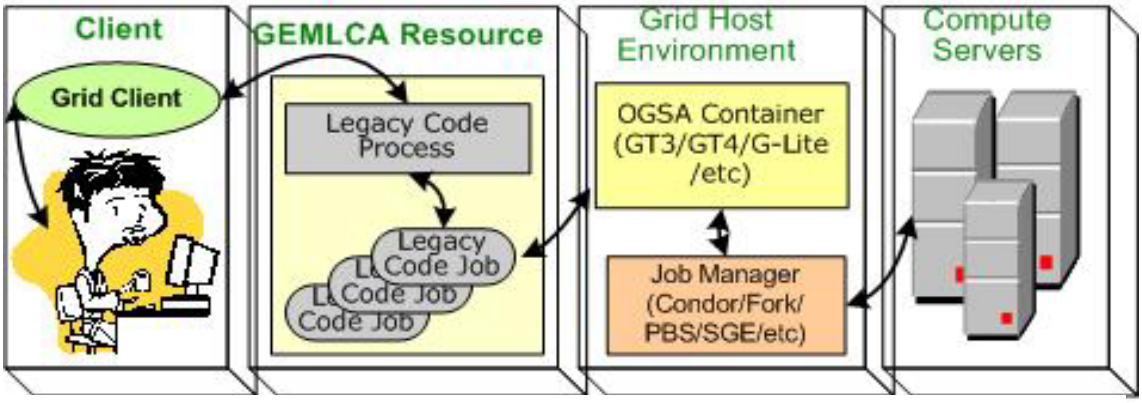


Fig. 1. GEMLCA conceptual architecture

tion. The deployment process of a GEMLCA 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 deployment defines the execution environment and the parameter set for the legacy application 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.

3. Integrating GEMLCA and the P-GRADE portal

GEMLCA provides the capability to convert legacy codes into Grid services just by describing the legacy parameters and environment values. However, an end-user without specialist computing skills still requires a user-friendly Web interface to access the GEMLCA functionalities: to deploy, execute and retrieve results from legacy applications. The P-GRADE portal offers these functionalities besides other capabilities like Grid certificate management, workflow creation, execution visualization and monitoring. This section describes how the integration enhanced the functionalities of both environments and how this integration can be even more effective in the future.

3.1 Extending the P-GRADE portal towards service oriented Grids

The P-GRADE portal supported only GT2 based Grids, originally. On the other hand, GEMLCA aims to expose legacy applications as GT3/GT4 Grid services. The integration of GEMLCA and the portal extended the GT2-based P-GRADE portal towards service oriented Grids. Users can still utilise GT2 resources through traditional job submission, and can also use GT3/GT4 resources by including GEMLCA

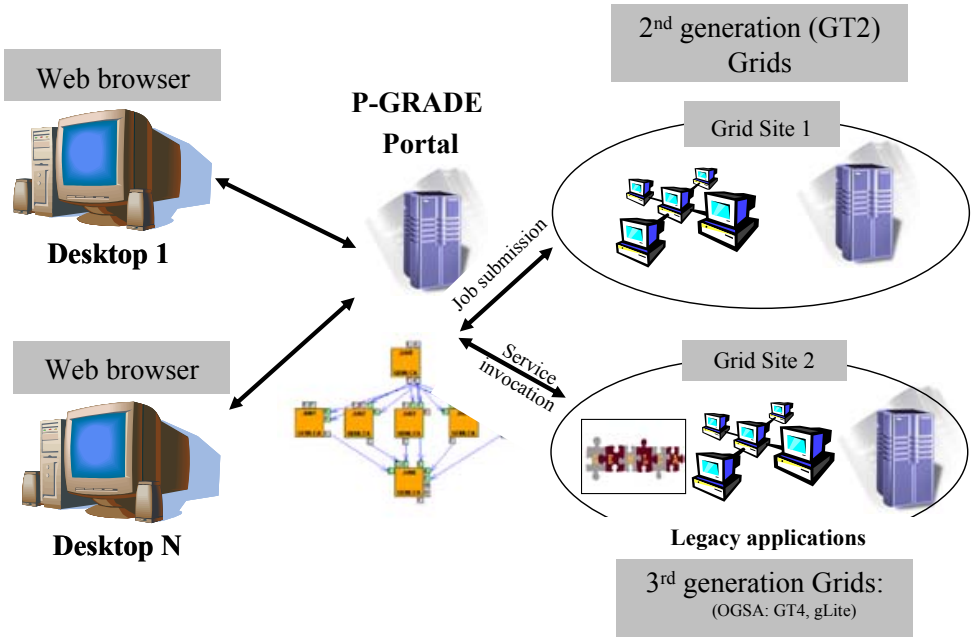


Fig. 2. GEMLCA extending the P-GRADE portal towards service oriented Grids

legacy code services in their workflows. The generic architecture of the GEMLCA – P-GRADE portal Grid is shown on figure 2.

Integrating the P-GRADE portal with GEMLCA required several modifications in the P-GRADE portal. These are as follows:

1. In the original P-GRADE portal a workflow component can be a sequential or MPI program. The portal was modified in order to include legacy code Grid service components as GEMLCA components.
2. The Job properties window of the P-GRADE portal was changed in order to extend it with the necessary legacy code support. The user can select a GEMLCA Grid resource from drop-down list. Once the Grid resource is selected the portal retrieves the list of legacy code services available on the selected Grid resource. Next, the user can choose a legacy code service from this list. Once the legacy code service is selected the portal fetches the parameter list belonging to the selected legacy code service with default parameter values. The user can either keep these values or modify them.
3. The P-GRADE portal was extended with the GEMLCA Administration Portlet that hides the syntax and structure of the LCID file from users. After filling in a simple Web form the LCID file is created automatically and uploaded by the portal to the appropriate directory of the GEMLCA resource.

After these modifications in the portal, end-users can easily construct workflow applications built from both GT2 jobs and legacy code services, and can map their execution to different Grid resources, as shown in Figure 3.

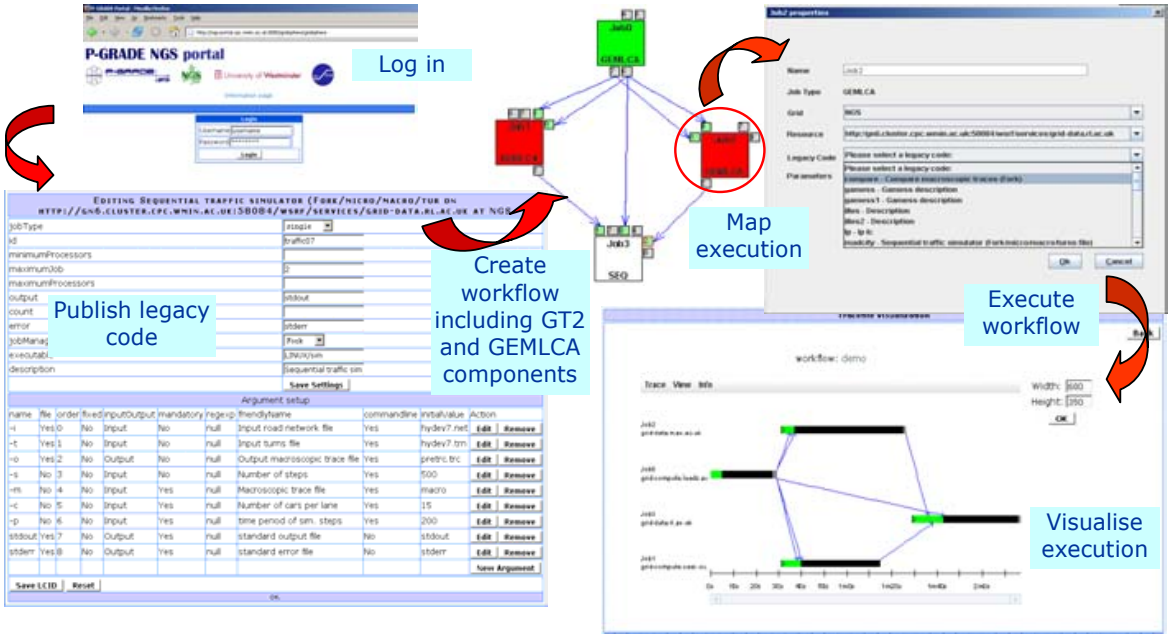


Fig. 3. Workflow creation, mapping and execution in the integrated GEMLCA – P-GRADE portal

3.2 Legacy code repository for production Grids

Creating a workflow in the P-GRADE portal requires the user to define input and output parameters and identify ports. For the owner of the code this task is not too complex. However, if another end-user wants to use the same code in a workflow the

process have to be repeated by a user who has no deeper understanding of the code itself. In order to help these end-users a legacy code repository based on GEMLCA was created that can be connected to GT2 based production Grid services, like the UK National Grid Service (NGS). The GEMLCA repository enables code owners to publish their applications as GEMLCA legacy codes in the same way as it was described in section 3.1. After this publication other authorised users can browse the repository and simply select the application they would like to use. Parameter values can be set by the user. However, there is no need to define parameter types or input/output ports as these are created automatically by the portal based on the GEMLCA description.

The P-GRADE portal extended with the GEMLA repository has been successfully implemented and offered for UK NGS users as a third party service [10]. This implementation of the integrated GEMLCA – P-GRADE portal solution extends the capabilities of both tools. On one hand, GEMLCA is now capable of working with GT2 based Grids by submitting the legacy executables as jobs to the remote GT2 gatekeeper. On the other hand, the usability of the P-GRADE portal has also been enhanced by making it much easier for end-users to create workflows using legacy codes published in the repository.

The integrated GEMLCA – PGRADE portal solution for GT2 based production Grids is shown of figure 4. The major challenge when connecting GEMLCA to the NGS was that NGS sites use GT2 however, the current GEMLCA implementations are based on service-oriented Grid middleware, namely GT3 and GT4. The interfacing between the different middleware platforms is supported by a script, called NGS script, that provides additional functionality required for executing legacy codes on NGS sites. To execute the code on a remote site first the NGS script, executed as a GEMLCA legacy code, instructs the portal to copy the binary and input files from the central repository to the NGS site. Next, the NGS script, using Condor-G, submits the legacy code as a job to the remote site.

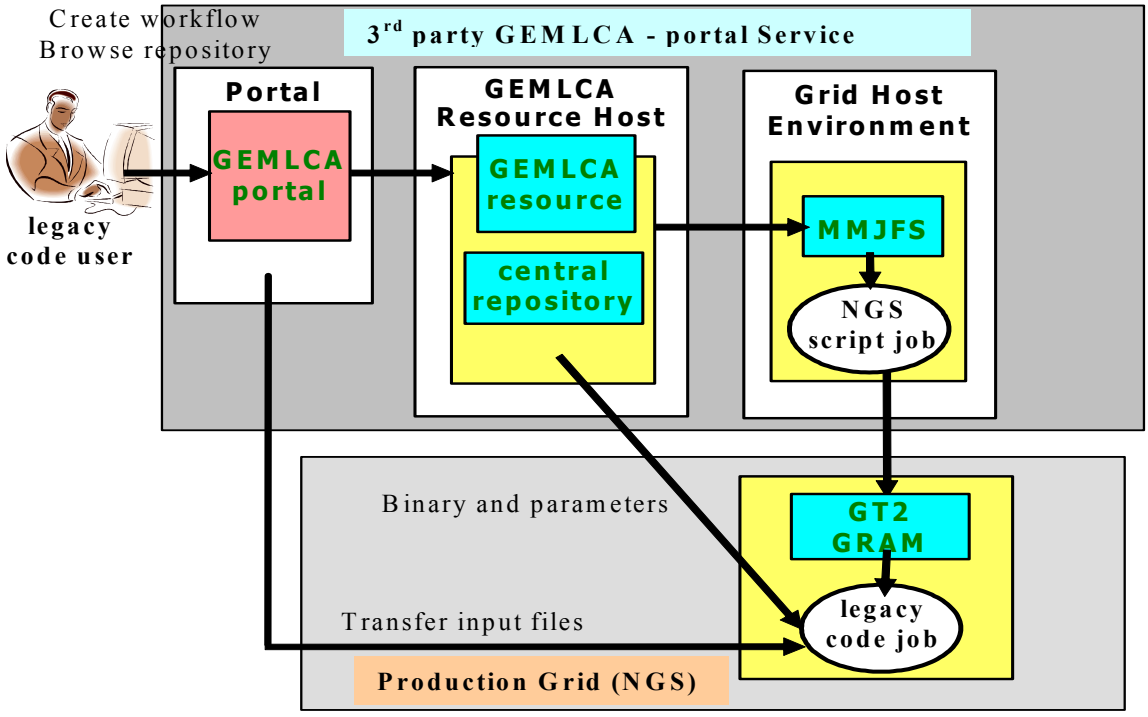


Fig. 4. GEMLCA and P-GRADE portal for GT2 based production Grids

3.3 Legacy Code Support for Desktop Grid Systems

The vision of Grid computing is to enable anyone to offer resources to be utilised by others via the network. This original aim, however, has not been fulfilled so far. Today's production Grid systems, like the EGEE Grid, the NorduGrid or the UK National Grid Service (NGS) apply very strict rules towards service providers, hence restricting the number of sites and resources in the Grid. The original aim of enabling anyone to join the Grid with one's resources has not been fulfilled. Nevertheless, anyone who is registered at the Certificate Authority of such a Grid and has a valid certificate can access the Grid and use the resources.

A complementary trend can also be observed for the other part of the original aim. Here, anyone can bring resources into the Grid system, offering them for the common goal of that Grid. Nonetheless, only some people can use those resources for computation. The most well-known example of these so called desktop Grid systems, is the SETI@home [9].

While the latest generation of traditional Grid systems are based on service-orientation as opposed to earlier resource-oriented models, today's desktop Grid systems do not follow this route. These Grids are more reminiscent of the traditional job submission model and resource-orientation, not allowing the invocation of pre-deployed services on workers. The work package executable has to be transferred to the worker and executed as a job.

The aim of our research in this area is to extend desktop Grids towards the service oriented concept and allow the desktop Grid master component to submit work packages to workers where the task is not represented by a downloaded executable but by a service invocation. The service-oriented model will enable desktop Grids to invoke and execute services on workers that can be either newly developed services or legacy applications presented as desktop Grid services.

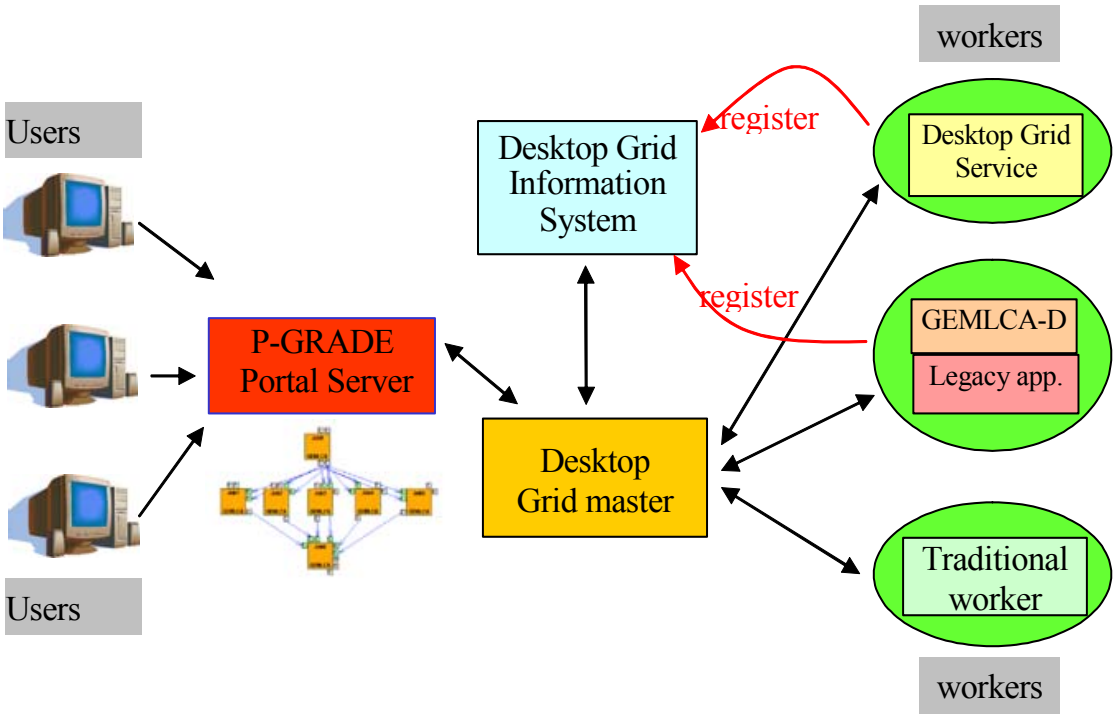


Fig. 5. GEMLCA and the P-GRADE portal for desktop Grids

In order to utilize services in a desktop Grid both the master and the client components has to be modified and extended with the service concept. The master has to invoke a service through a standard protocol. The client should know that it has to start a service, instead of a downloaded binary executable, and has to pass and, if necessary, convert parameter values for this service. The master has to be aware of the deployed services and their required parameter values. An information system (see figure 5) is required where services can be registered and discovered by the master.

Besides newly developed services it is very important to incorporate existing legacy code applications into the desktop Grid architecture. There are existing programs that could be useful to utilize in desktop Grid systems as services in order to extend their usability and the class of applications these Grids could be applied for. However, these legacy applications were not specifically designed as desktop Grid services, and cannot be directly connected to the architecture. Moreover, the deployment of these applications could be rather demanding due to the intensive use of external libraries and environmental dependencies, and could not be fulfilled by simply downloading the binary executable.

To achieve these objectives the integrated GEMLCA P-GRADE portal environment is migrated to desktop Grid architectures enabling the utilisation of both newly developed and legacy code services in desktop Grid workflows significantly extending the usability of desktop Grids. In order to utilise the GEMLCA concept for desktop Grids significant modification of the architecture is required. Some aspects of the GEMLCA solution, like the service description or the XML based Legacy Code Interface Description format, are immediately applicable for desktop Grids. However, the job submission and execution mechanisms that currently use Globus MMJFS (Master Managed Job Factory Service) have to be redesigned and significantly modified. The resulting solution, GEMLCA-D enables the deployment of legacy code services on desktop Grids without code re-engineering. The structure of this extended desktop Grid architecture is illustrated on figure 5.

3.4 GEMLCA as a plug-in in the P-GRADE portal

Both GEMLCA and the P-GRADE portal are continuously developing products. In order to present their latest features in a uniform way, the two systems must be integrated into a single software from time to time. Although the integration could be done with reasonable effort in case of the first few GEMLCA and portal releases, the task became quite cumbersome with the appearance of additional features and grid middleware technologies.

The P-GRADE portal and the GEMLCA developer teams elaborated and are currently working on the implementation of a plug-in based concept to handle this issue. According to the idea the portal will be transformed into a component-based environment that can be extended with dynamically downloadable plug-ins on the fly. The approach exploits the fact that GEMLCA and P-GRADE jobs are quite similar to each other. They both consist of a binary executable and 1-N input files, a set of input parameters, a resource that has been selected for the execution, etc. The only differ-

ence is that the executable and the input files of a GEMLCA job can come from different users, the executable and the input files of a P-GRADE job must be provided by the same party. Nevertheless, this difference can be hidden from the higher software layers during execution by the workflow manager subsystem (an intelligent script can transfer executable and input files to the executor site uniformly, even if those files are coming from different parties), due to the different concept GEMLCA and P-GRADE job developers must be provided with different user interfaces. (As it was described in Section 3.1 other types of parameters must be set on the "Properties" windows for a GEMLCA and for a P-GRADE job.)

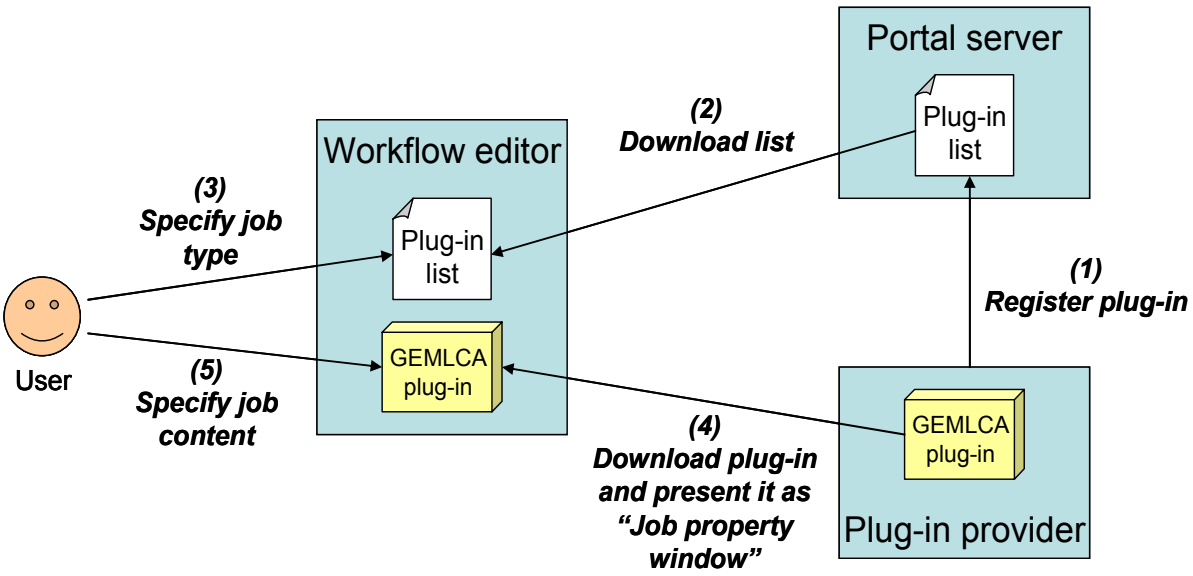


Fig. 6. The plug-in based P-GRADE portal workflow editor

While job property windows are hard coded parts of the current version of the workflow editor, the new plug-in based concept enables the dynamic download of these graphical elements, making the P-GRADE and GEMLCA integration task almost self evident. The plug-in based editor works in the following way: (See also Figure 6) When the user drops a new job onto the editor canvas and opens its property window, the editor obtains a list of the currently available job types from the portal server (2). Each entry of this list consists of a name (e.g. GEMLCA job) and a reference pointing to a dynamically downloadable plug-in. In subject to the choice of the user (3) the editor downloads the plug-in that belongs to the selected job type and presents it on its GUI as the content of the job property window (4). Since each plug-in can encapsulate customised graphical elements to support the development of one specific type of job (5), no property windows must be hard coded into the workflow editor any more.

Because the workflow editor is a Web Start application plug-ins can be implemented as Java objects. While the list of plug-ins can be retrieved from the portal server using a text format (e.g. an XML message), a binary protocol is needed between the editor and the plug-in provider to make object transmission possible. Java Web Services [6], Jini services [5], EJBs [7] or RMI servers [8] are all suitable for this purpose, thus the plug-in provider can be implemented with any of these technologies. Besides GEMLCA jobs other types of computational jobs can be plugged

into the portal in this way. The plug-in provider has to only register its service at the portal server (1).

4. Conclusions and future work

With the integration of the GEMLCA and the P-GRADE portal tools a complex environment has been created that provides solution for a wide range of grid-related problems. Using the integrated system Grid users can deploy legacy and newly developed sequential and parallel applications as software services. They can test and then share these services with a larger community. The members of the community can develop workflows that connect their own computational tasks and the predeployed services into an acyclic graph. These workflows can be submitted into Globus-2, 3 and 4 based Grids, can be monitored in a real-time fashion, moreover, the different components can be executed in different Globus VOs.

As the next step, the architecture will be ported to desktop Grid systems in order to extend their functionality with legacy service support. Also, the P-GRADE portal will be transformed into a plug-in based computational framework. With the plug-in concept the portal will be able to support other types of computational services than GEMLCA, without modifying any source code or even stopping the portal server.

References

[1] G. Sipos and P. Kacsuk: Classification and Implementations of Workflow-Oriented Grid Portals, To appear in the Proc. of The 2005 International Conference on High Performance Computing and Communications (HPCC2005), Sorrento, Italy

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

[3] T. Tannenbaum, D. Wright, K. Miller, and M. Livny: Condor - A Distributed Job Scheduler. Beowulf Cluster Computing with Linux, The MIT Press, MA, USA, 2002.

[4] I. Foster, C. Kesselman: Globus: A Toolkit-Based Grid Architecture, In I. Foster, C. Kesselmann (eds.) „The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 1999, pp. 259-278.

[5] J. Waldo. The Jini architecture for network-centric computing.*Communications of the ACM*, 42(10):76–82, Oct. 1999.

[6] D. Chappell and T. Jewell, "Java Web Services", O'Reilly Press, 2002.

[7] Sun Microsystems: Enterprise JavaBeans: http://java.sun.com

[8] Troy Bryan Downing. *Java RMI: Remote Method Invocation*. Number 0764580434. IDG Books, 1998.

[9] D.P. Anderson, J. Cobb, E. Korpela,M. Lebofsky,D.Werthimer. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, Vol. 45 No. 11, November 2002, pp. 56-61

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

GRIDLE Search for the Fractal Component Model

Diego Puppin[1], Matthieu Morel[2], Denis Caromel[2],
Domenico Laforenza[1] and Françoise Baude[2]

[1] ISTI-CNR
via Moruzzi 1, 56100 Pisa, Italy
{diego.puppin, domenico.laforenza}@isti.cnr.it
[2] INRIA
2004 route des lucioles - BP 93, FR-06902 Sophia Antipolis, France
{matthieu.morel, denis.caromel, francoise.baude}@sophia.inria.fr

Abstract. In this contribution, we discuss about the features needed by a component-oriented framework, in order to implement a tool for component search. In particular, we address the Fractal framework: we describe its main features and what is needed to perform effective component search in this framework.

1 Introduction

There is a growing attention to component-oriented programming for the Grid. Under this vision, complex Grid applications can be built simply by assembling together existing software solutions. Nonetheless, two main features are still missing: a a standard for describing components and their interactions, and a service able to locate relevant components within the Grid. An important step in the first direction was taken by adopting Fractal as a base for the component model for the CoreGrid project. The use of a common component model will accelerate the convergence of several initiatives in the Grid. Still, the research in tools for an effective component search is lagging behind.

This contribution is structured as follows. After an overview of relevant work in the field of component search and ranking (Section 2), we introduce our vision of a search engine based on the concept of component ecosystem (Section 3). Then, we describe the Fractal framework (Section 4) and the features we need to implement our search strategy over it (Section 6). Finally, we conclude.

2 Related work

Valid ranking metrics are fundamental in order to have relevant search results out of the pool of known components. The social structure of the component ecosystem is, in our opinion, the strongest source of information about component relevance: components that are used by many other trusted components are probably *better* than those that are rarely used. Below, we illustrate different approaches to ranking discussed in the literature.

In [3], the authors cite an interesting technique for ranking components within a set of given programs. Ranking a collection of components simply consists of finding an absolute ordering according to the relative importance of components. The method followed by the authors is very similar to the method used by the Google search engine to rank Web pages: PageRank [1]. In ComponentRank, in fact, the importance of a component is measured on the basis of the number of references (imports, and method calls) other classes make to it within the given source code. Unfortunately, ComponentRank is designed to work with isolated software projects, and it is not able to grasp social relations among independently developed components as we do.

There has been a number of interesting works also in the field of *workflow mining*. The approach we propose is complementary to that presented in [5]: the dynamic realization of a workflow is analyzed in order to discover profiling properties, frequently used activities and so on. This analysis requires access to the actual workflow execution, which is usually kept secret by the application providers. We believe that an approach focused on the static structure of the workflow has a stronger potential.

Another related result was recently presented by Potanin et al. [4]. The authors examined the heap of large Java programs in order to measure the number of incoming and outgoing references relative to each object in the program. They verified that the number of references is distributed according to a power-law curve w.r.t. the rank. Their work is based on the *dynamic* realization of a program. We show that this relationship exists also in the *static* links among classes in their coding.

3 The GRIDLE Search Framework

We would like to approach the problem of searching software components using the mature Web search technology. In our vision, an effective searching and ranking algorithm has to exploit the *interlinked structure* of components and compositions. Our ranking scheme, in fact, will be aware of the context where components are placed, how much and by who they are used.

Figure 1 shows the overall architecture of our Component Search Engine called *GRIDLE*: *Google^{TM}-like Ranking, Indexing and Discovery service for a Link-based Eco-system of software components*. The main modules of GRIDLE are the *Component Crawler*, the *Indexer* and the *Query Answering*.

The *Component Crawler* module is responsible for automatically retrieving new components. The *Indexer* has to build the *index* data structure of GRIDLE. This step is very important, because some information about the relevance of the components within the ecosystem must be discovered and stored in the index. The last module of GRIDLE is the *Query Answering* module, which actually resolves the queries on the basis of the index.

In the next paragraph, we discuss some of the properties of the social network of components, and the way we used it to perform ranking.

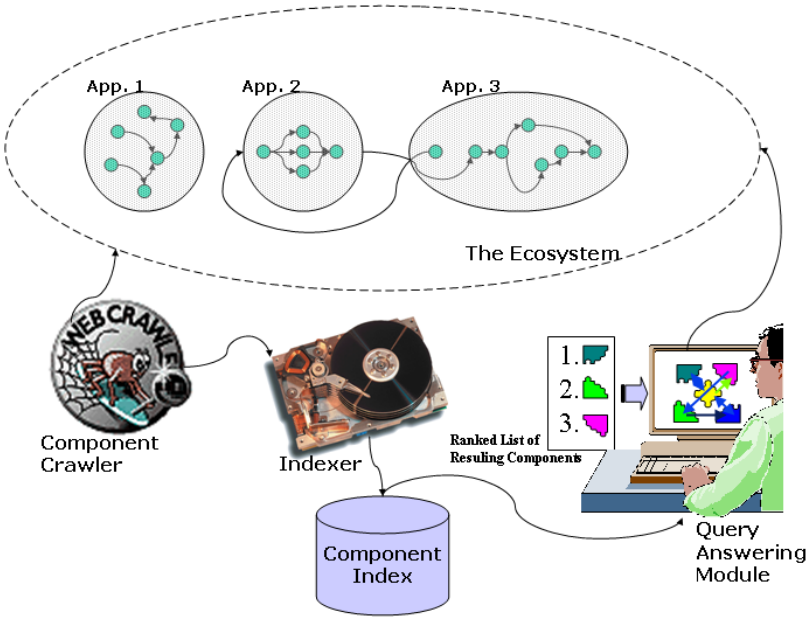


Fig. 1. The architecture of GRIDLE.

3.1 Component Ecosystem: Experiments

In order to explore the characteristic of the component ecosystem, we performed some initial experiments on Java classes. We were able to collect a very large number of Java classes from around the Internet. Clearly, Java classes are only a very simplified model of software components, because they are supported by only one language and they cannot cooperate with software developed with other languages, but they also support some very important features:

1. their interface can be easily manipulated by introspection;
2. they are easily distributed in form of single JAR files;
3. they have a very consistent documentation in the form of JavaDocs;
4. they can be manipulated visually in some IDEs (BeanBox, BeanBuilder etc.);
5. they have a natural unique ID (given by the package name);
6. it is easy to see which class uses which others.

In our experiments, we looked for *documented* Java classes, which can be used by independent developers in their programming tasks: our target were all Java classes with consistent Java Documentation files (JavaDocs). We search the Web looking for JavaDocs. Starting points of our searching were those documents reached through standard web search engine and the documentation site of projects we were aware of. Within the collected JavaDocs, we also found links to external libraries used by the developers.

This way we collected an initial set of 7700 classes, then grown to 49500. We were able to retrieve very high-quality JavaDocs for big software projects, including: Java 1.4.2 API; HTML Parser 1.5; Apache Struts; Globus 3.9.3 MDS; Globus 3.9.3 Core and Tools; Tomcat Catalina; JavaDesktop 0.5, JXTA; Apache

Lucene; Apache Tomcat 4.0; Apache Jasper; Java 2 HTML; DBXML; ANT; Nutch; Proactive; Fractal; Eclipse.

We parsed the JavaDocs files, and we recorded a link between Class A[3] and Class B everytime a method in Class A used as an argument, returned as a result, or listed as a field, an object of type B. This way, we generated a directed graph describing the social network of the Java libraries.

The basic assumption behind this is that a Java programmer behaves somewhat like a general service user: s/he will pick up the most useful service (Java class) out of a large repository of competing vendors/providers (libraries). Thus, s/he will have an eye to the most useful and general classes.

We then counted and plotted the number of inlinks and outlinks from each class. We observed a very interesting power-law distribution (see Figure 2): the number of inlinks, i.e. the number of times different classes refer each class, is distributed following a power-law pattern: very few classes are linked by very many others, while several classes are linked by only a few other classes. This is true both for our initial sample of 7700, and for the bigger collection.

In the figures, the reader can see a graph representing the number of incoming links to each class, in log-log scale. Classes are sorted by the number of incoming links. The distribution follows closely a power-law pattern, a small exception given by the first few classes (Object, Class etc.) which are used by almost all other derived classes to provide basic services, including introspection and serialization.

This is a very interesting result: within Java, the popularity of a class among programmers seems to follow the pattern of popularity shown by the Web, blogs and so on (see [2]). This supports our thesis that methods for Web search can be used effectively also for component search.

3.2 GRIDLE 0.1: Searching Java Classes

Out of our preliminary results, we developed a prototype search engine, able to find high-relevance classes out of our repository. Classes matching the query can be ranked by TF.IDF (a common information retrieval method, based on a metric that keeps into account both the number of occurencies of a term within each document and the number of documents in which the term itself appears) or by GRIDLERank, our version of PageRank for Java classes, based on the class usage links. Our first implementation is available on-line at: http://gridle.isti.cnr.it/.

GRIDLERank is a very simple algorithm, that builds on the popular PageRank algorithm used in Google[1]. To determine the rank of a class C, we iterate the following formula:

$$rank_C = \lambda + (1 - \lambda) \sum_{i \in inlinks_C} \frac{rank_i}{\#outlinks_i}$$

where $inlinks_C$ is the set of classes that use C (with a link into C), $\#outlinks_i$ is the number of classes used by i (number of links out of i), and λ a small factor, usually around 0.15.

[3] We added the Interface files to our collection of Classes.

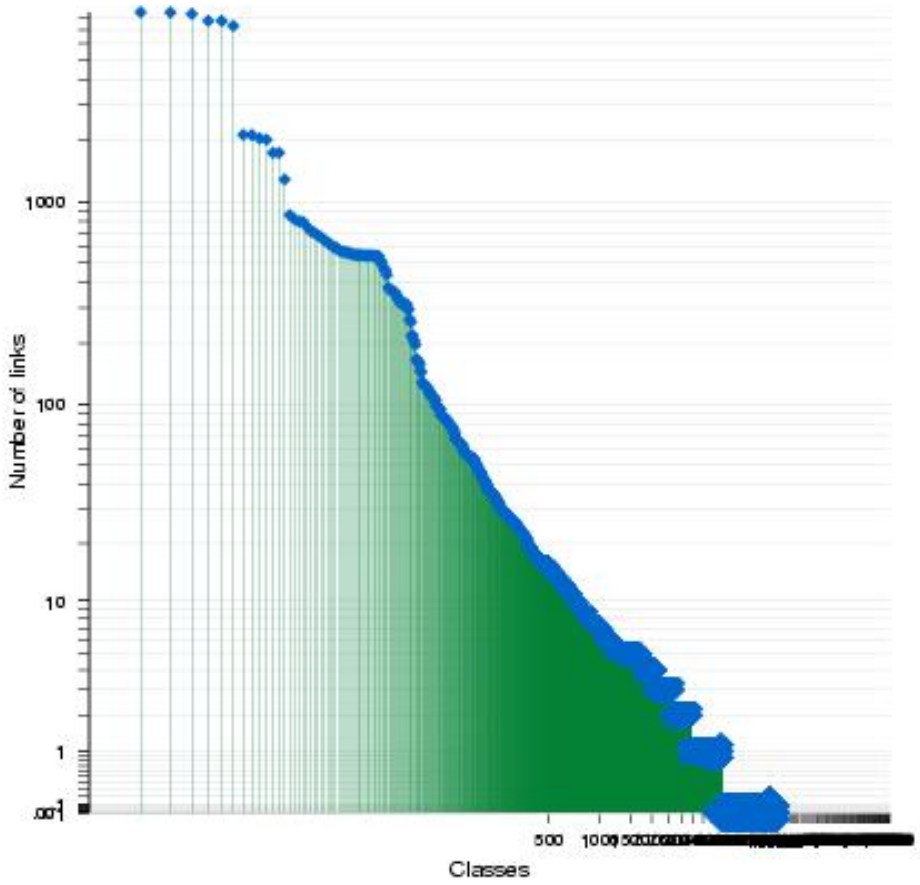


Fig. 2. Distribution of inlinks, 49500 classes.

Interesting Observations We could observe some very interesting results. The highest-ranking classes are clearly some basic Java API classes, such as `String`, `Object` and `Exception`. Nonetheless, classes from other projects are apparently very popular among developers: #7 is Apache `MessageResources`, #11 is Tomcat `CharChunk`, #14 is DBXML `Value` and #73 is JXTA `ID`. These classes are very general, and are used by developers of unrelated applications.

We could verify that in most cases GRIDLERank is more revelant than TD.IDF, especially when the class name is not textually similar to the function we are looking for. For instance, if the developer is trying to write data to a file, and performs a query such as *"file writer"*, TF.IDF will return, in order: (1) javax.jnlp.JNLPRandomAccessFile, from JNLP API Reference 1.5; (2) javax.swing.filechooser.FileSystemView, from Java 2 Platform SE 5.0; (3) java.io.FileOutputStream, from Java 2 Platform SE 5.0; (4) java.io.RandomAccessFile, from Java 2 Platform SE 5.0. The second class is clearly unrelated with the problem under analysis, and only the third is probably what the user was looking for.

On the other hand, GRIDLERank will return four classes from the Java API (Java 2 Platform SE 5.0): (1) java.io.PrintWriter; (2) java.io.PrintStream; (3) java.io.File; (4) java.util.Formatter; which are all probably better matches. To

verify this claim on result quality, we will need to test the search engine with Java developers.

4 Fractal and the Grid

4.1 The Fractal Component Model

Fractal is a simple, flexible and extensible component model developed by the ObjectWeb consortium. It 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 main design goal of Fractal was to reduce the costs of developing and maintaining big software projects, in particular the ObjectWeb projects. The Fractal model already uses some well known design patterns, such as separation of interface and implementation and, more generally, separation of concerns, in order to achieve this goal.

Recently (June 2005), Fractal was chosen as a basis for the component model for the CoreGrid network. As a matter of fact, Fractal has several features that are fundamental for Grid programming.

- Several different communication patterns can be implemented using Fractal, so to cope with communication latency, including asynchronous communication, subscribe/pulling.
- It allows structural and hierarchical composition, which is a fundamental requirement to build more and more complex applications.
- Sub-components can be shared among components, this way implementing a coherent single-state image of the system.
- Components can be reconfigured, and external controllers can be used to wrap a component.

Currently, the Fractal project is a big effort, organized into four sub projects:

1 The Component Model sub project deals with the definition of the component model. The main characteristics of this model are recursivity (components can be nested in composite components - hence the "Fractal" name) and reflexivity (components have full introspection and intercession capabilities). The Fractal model is also language independent, and fully modular and extensible.
2 The Implementations sub project deals with the implementation of Fractal component platforms, which allow the creation, configuration and reconfiguration of Fractal components. Julia, the reference implementation, is developped in this sub project.
3 The Components sub project deals with the implementation of reusable, ready to use Fractal components, such as protocol or Swing components.
4 The Tools sub project deals with the implementation of Fractal based applications dedicated to Fractal, such as tools to define component configurations.

One of the core principle behind Fractal is the so-called *separation of concerns*, this means the need to separate, into distinct pieces of code or runtime entities, the various concerns or aspects of an application: implementing the service provided by the application, but also making the application configurable, secure, available etc. There are three main aspects:

1. Separation of interface and implementation.
2. Component oriented programming.
3. Inversion of control.

The first refers to need to separate the design and implementation concerns. A component must stick to a precisely defined *contract*, at syntactic and semantic level. The component contracts must be designed with care, so as to be the most stable as possible w.r.t. internal implementation changes: interfaces must deal only with the services provided by the components — not implementation or configuration of the components. Configuration concerns are to be be dealt with separately, as well as other concerns such as security, life cycle, transactions...

The second means the separation of the implementation concern into several composable, smaller concerns, implemented in well separated entities called components. Wrapper components can add levels of security, authentication, reconfiguration etc. A wrapper component can take an existing component, wrap it, and export the interface to a similar component, with added features. For instance, a wrapper component can take a sequential component and launch it in another thread, offering explicit control of its execution (start, stop etc.).

As an implementation optimization, wrappers can be implemented *inline* by rewriting the Java bytecode. Julia (a Java implementation of Fractal) offers life-cycle controllers and other wrappers in the form of code generators.

The third principle, *inversion of control*, is the separation of the functional and configuration concerns. Components are configured and deployed by an external, separated entity. This way, components can be replaced, updated etc. *at run-time*, not only at design-time.

In other words, a Fractal component is indeed composed of two parts:

1. a content that manages the functional concerns,
2. a controller that manages zero or more non functional concerns (introspection, configuration, security, transactions, ...).

The content is made of other Fractal components, i.e. Fractal components can be nested (and shared) at arbitrary levels.

In order to be used with the framework, a component must stick to a set of design conventions. In particular, within the Julia implementation, it has to implement a set of Java interfaces with the methods for binding other components, export the interface and so on (see Table 1).

When this standard is respected, components can be easily manipulated within the included graphical shell (see Figure 3). The recursive structure of composition is clearly visible.

```
public interface Component {
   Object[] getFcInterfaces ();
   Object getFcInterface (String itfName);
   Type getFcType ();
}
public interface ContentController {
   Object[] getFcInternalInterfaces ();
   Object getFcInterfaceInterface(String itfName);
   Component[] getFcSubComponents ();
   void addFcSubComponent (Component c);
   void removeFcSubComponent(Component c);
}
```

Table 1. Basic methods to be implemented by a Fractal component within the Julia framework (Java implementation of Fractal).

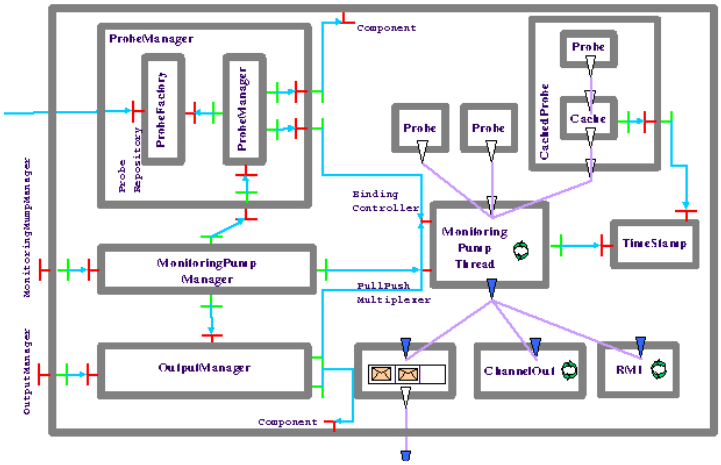


Fig. 3. The logical structure of assembled Fractal components. Manipulation within the graphical environment (screen snapshot).

4.2 Fractal and Proactive

Recently, the Fractal framework was ported to Proactive. This way, Fractal components can be also active objects, as described by the Proactive framework. Components can so migrate from one machine to another, can use asynchronous communication and so on. The Proactive framework itself is currently being re-factored to stick to the Fractal specification.

5 Needed Features

In this section, we discuss the features we need to implement our component search for Fractal components. We believe that some crucial tool and standard are still missing.

5.1 Documentation

As described, GRIDLE bases its analysis on the application structure and on free-text search among documentation files. With these two requirements, GRIDLE can select the components that match a given query and arrange the results according to our ranking metrics.

The current Fractal model does not include a strong standard for documentation file nor a way to distribute them: currently, only interfaces have a standard description with their ADL file.

We analyzed some Java application developed using the Fractal framework (including the reference implementation of the framework itself). In all cases, the detailed description of the interfaces (expected behavior etc.) is usually released in the form of a Java documentation file for the corresponding Java interface.

This will allow us to implement a search tool that chooses among different types of services, but not to choose among different service providers (e.g. different component implementations).

When more detailed information about a given implementation is needed, there is no clear standard for composite components, while, for basic components, we can observe a slight confusion among Fractal components and Java classes: the Java classes implementing the components are described with the standard JavaDoc files. This is not satisfactory for several reasons.

1. Methods and fields that are peculiar to the framework (e.g. the *bindFc* mechanism to bind components) are listed among the methods that implements the class logic.
2. Component dependencies (client interfaces) to other components can be inferred only through a detailed analysis of class fields, among which the binding variables are listed.
3. Redundant information is present about, for instance, methods inherited through the Java class structure (e.g. *clone, equals, finalize, getClass, toString...*) or interfaces implemented for other reasons related to implementation details (e.g. *javax.swing.Action, java.lang.Cloneable, java.io.Serializable...*).

This can be partly solved by browsing the interface documentation but it is not satisfactory, as this mixes the Java implementation with the component structure.

5.2 Packaging

A standard for component distribution and management is not yet part of the Fractal model. There is an on-going effort, within the developers' community, to include some packaging tools within the Fractal framework. An experimental tool, called FractalJAR, is currently under development.

A packaging standard should describe a way to ship components in a compact form (single-file archive), including:

– ADL files describing the internal composition of the component;

– implementation (executable) files;
– description files, with clear interface, dependency and behavior description.

Also, the package should have a strong version system, which should manage dependencies with different component versions and component upgrade. It should verify that an update does not break the coherence of the software infrastructure.

As a matter of fact, an interesting goal of searching would be to find complete packaged components (compositions), rather than individual components with outstanding dependencies.

Packaged components should also be easier to store and archive in specific repositories.

6 Conclusion

In this contribution, we presented our vision of a tool searching software components, and we discussed how to adapt it to Fractal.

We believe that in the near future there will be a growing demand for ready-made software services, and current Web Search technologies will help in the deployment of effective solutions.

When all these services become available, building a Grid application will become a easier process. A non-expert user, helped by a graphical environment, will give a high-level description of the desired operations, which will be found, and possibly paid for, out of a quickly evolving market of services. At that point, the whole Grid will become as a virtual machine, tapping the power of a vast numbers of resources.

References

1. S. Brin and L. Page. The Anatomy of a Large–Scale Hypertextual Web Search Engine. In *Proceedings of the WWW7 conference / Computer Networks*, volume 1–7, pages 107–117, April 1998.
2. Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *Proceedings of SIGCOMM*, 1999.
3. Katsuro Inoue, Reishi Yokomori, Hikaru Fujiwara, Tetsuo Yamamoto, Makoto Matsushita, and Shinji Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th international conference on Software engineering*, pages 14–24, Portland, Oregon, May 2003. IEEE, IEEE Computer Society.
4. Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in oo programs. *Communications of the ACM*, 48:99–103, May 2005.
5. W.M.P. van der Aalst and B.F. van Dongen and J. Herbst and L. Maruster and G. Schimm and A.J.M.M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47:237–267, 2003.

Grid computing performance prediction based in historical information[1]

Francesc Guim[1], Ariel Goyeneche[2],
Julita Corbalan[1], Jesus Labarta[1], Gabor Terstyansky[2]

[1] Computer Architecture Department, Universitat Politècnica de Catalunya,
{fguim,juli,jesus}@ac.upc.edu
[2] Centre for Parallel Computing, Cavendish School of Computer Science,
University of Westminster,115 New Cavendish Street, London, W1W 6UW
{goyenea,terstyg}@wmin.ac.uk

Abstract. Performance prediction in Grid computing presents an important challenge due to Grid environments are volatiles, heterogeneous and not reliable. We suggest that the historical data related to applications, resources and users can provide an adequate amount of information for modelling and predicting Grid components behaviours. Such information can be used to build a dynamic top-bottom Grid Service and Resources performance descriptions. In this paper we present the initial work that we have done in order to define prediction techniques. We show how the analysis of the CEPBA-UPC centre workload has guided us in the design of seven different prediction techniques. We present the mechanism used for the performance evaluation of all the described predictors and their analysis.

1 Introduction

Performance refers to system responsiveness: either the time required to respond to specific events, or the number of events processed in a given time interval. For traditional information systems, performance considerations are often associated with usability issues such as response time for user transactions. On the other hand, the performance targets in Grid computing has had to be adapted to support heterogeneous and not reliable environments, where the analysis information can be limited, incomplete and non up-to-date.

Currently, at best, computational scientists, researchers and HPC users access Grid Services and Resources via Grid portals, which provide common components that can

[1] The integration work published in this paper has been supported by "*CoreGrid*", network of excellence in "*Foundations, Software Infrastructures and Applications for large scale distributed, Grid and Peer-to-Peer Technologies*, and by the Spanish Ministry of Science and Education under contract TIN2004-07739-C02-01.

securely help them to create, submit, and schedule Grid jobs. In addition, portal users may perhaps be allowed to track and monitor submitted jobs. However they still do not have the essential information in order to identify in advance the job performance behaviours and possible execution cost. At the Grid infrastructure level, performance prediction is fundamental for Scheduler and Brokers, which can improve their suggestions and decision having an approximate picture of the applications behaviour.

In order to answer some of the issues related to the Grid computing performance prediction area, we suggest that the historical data related to applications, resources and users can provide an adequate amount of information for modelling and predicting its behaviours. As we are aware of the complexity of this topic and the added difficulty of working with real workloads, we decided to carry out a preliminary stage that could guide us on the processes of designing such techniques. Mainly, the objectives of this stage are to understand the behaviours and characteristics of submitted jobs and to design and test performance prediction techniques.

The remainder of the paper is organized as follows. Section 2 spots performance in Grid computing, Section 3 describes an approach of solving this issue, Section 4 tackles the paper's objectives by presenting general mechanisms in order to evaluate predictors and comparing their performance, designed predictors and analysis of their results. Finally, the current work conclusion and future work is described.

2 Performance in Grid Computing

Traditional computing performance research is focused in the exact qualification of software programs and resources, and an accurate prediction of its run-time performance. On the other hand, performance in Grid computing could be defined as the source of information to identify the most reliable source of capacity computing power.

The open question that needs to be addressed is how the performance knowledge, that has been a topic of much scrutiny for an important number of years, could be adapted to reflect the grid resources and what has to be changed, improved and added to fulfil these new characteristics.

2.1 Filling a vacuum

A snapshot of the most important traditional performance techniques may start with *execution-driven* [1] performance simulators. They can be acutely dismissed for Grid computing on the grounds of being extremely slow, and memory intensive. Techniques using *trace-driven* [2] performance analysis, where the inputs to test programs and resources are traces of dynamic instructions without full-function, are not an option given that the complexity of microarchitecture has increased dramatically, causing *trace-driven* performance simulation to become time consuming and, if having any result, too resource related. Also, *instruction-level* [3] methods that conceptually simulate all feasible paths on arbitrarily detailed timing models of the hardware plat-

form, assumes that input data is known and therefore only analyze a single path through the program associated with this input data.

Although there are examples [4] [5] [6] that try to improve some of the previous mentioned problems, all of them results in being too machine specific and non dynamic for Grid environments.

Consequently, performance in a Grid computing has to be considered using a different approach: Our strategy aims to exercise a *top-bottom* self learning performance description of Grid Resources and Services that can be used to determine performance predictions. The dataset is build from the historical usability information of Grid resources, services.

The *top-bottom* means from the very simple Grid Services and Resources description towards a more complex description that reflects its compositions and details. The axiom of this idea may be seen as "The more Resources and Services are used, the betted and more accurate the descriptive information should be". This practice aspires to keep always learning and converging, given the volatile aspect of Grid environments, the best confluence of data descriptive structures, metadata, and granularity for Grid components that provides the best analysis for performance prediction.

3 Grid Performance description models

The description of Grid services, resources and run-time environment plays a key factor in our strategy. Many of the few existent Grid performance technologies use Software Performance Engineering (SPE) [7] [8] to provide a representation of the whole system. SPE is the systematic process for planning and evaluating a new system's performance throughout the life cycle of its development. Its goals are to enhance the responsiveness and usability of systems while preserving quality. SPE includes techniques for gathering data, coping with uncertainty, constructing and evaluating performance models, evaluating alternatives, and verifying models and validating results. It also includes strategies for the effective use of these techniques.

SPE needs a number of different pieces of information in order to construct and evaluate early life-cycle performance models. These information requirements fall into the following categories: workload, performance objectives, software characteristics and information model, execution environment, resource requirements and processing overhead

3.1 SPE and Grid Computing

Taking into consideration the Grid computing scenario, there are some points to be considered at the time of using SPE in Grid Computing:

– The definition of software and system execution models is rather difficult because of the dynamicity of the system and rights to access them.

– The complexity of accurate performance measures may significant given that SPE is pointed to address performance information precision when the models granularity is small.
– And, for some performance computing system, which involve concurrency and parallelism, the models is not completed and must be enhanced [9].

Performance Analysis and Characterization Environment (PACE) [10] is an example of the SPE enhanced and used in Grid computing by the inclusion of a parallel layer between the hardware and the application model. PACE can be used to produce predictive traces representing the expected execution behaviour of an application given appropriate workload information and has the motivation to provide quantitative data concerning the performance of sophisticated applications.

But, because PACE is a solution that has been migrated and afterwards extended from a traditional performance computing research to a Grid environment, it assumes several restricted points, such us the need of the source code, which has to be written in language C, of each software component in the Grid and the requirement of running the Resource tools in each new or modified Grid resource in order to generate the hardware model

Consequently, this example emphasize the idea of a Grid SPE that is able to, dynamically, auto-learn and consolidate in a Grid performance description model, which initially is contemporary to the environment where the information is coming from, and afterwards, this model could be enhanced in order to extrapolate it, by projecting the information, to other particular systems

3.2 Grid SPE: Performance Meta-data models

Therefore, the very first meta-data classification could be defined by the use of a fixed-scenario, based in a particular run in a particular scenario in a particular run-time environment or, altematively, analytical approaches that can produce parametric models.

The performance model granularity and dynamism restricts the approach to collect and use meta-data information. A model that adds on top of a basic structure a statically meta-data that helps analytical approaches could be used in both scenarios. But the use of static structures in volatile environment produces a non up-to-date model that constantly increases in volume and complexity. As a result, the model has to use the meta-data information as a feedback to dynamically upgrade the description model.

At this point, there are several issues that have to be considered in order to reach the proposed approach, such as:

– How important a SPE component of a Grid Service or Resource Description Model is?
– Meta-data weight: Is it possible to assign a level of importance to the data collected from the Grid? Is this data influenced by the SPE component weight?
– Is the Meta-data historical information depreciated by time?
– It is also depreciated by Resource or Service variations?
– How much influence the time that takes to process the information could affect the model?

– How can we quantify the error of prediction?

In order to start tackling these open questions, in the following section an initial work done on prediction techniques is presented. We have focused on establishing mechanisms for designing and evaluating predictors in order to understand, how all the components in Grid computing can be put together to solve performance problems. We start testing simple prediction techniques based on historical data. Before design complex techniques, we studied simple predictors that try to exploit the idea that user and groups use behave similar in their executions.

4 Grid Performance predictors

Initially this section presents a global definition of the predictor performance, the second part presents in details each of them and their main objectives, and finally the performance evaluation of such predictors is described.

4.1 Evaluating the predictors

There are two different issues that have to be treated: the first one is to decide when a given prediction is a *hit* or a *miss*, and more important how to define the global performance of a given predictor with a set of predictions.

4.1.1 Hit / Miss definition:

Definition of what is a hit or what is a miss can be seen on the formula picture below(**1**). The definition of a hit or miss is based on the difference between the real and the predicted value: if the difference is less than 5% of the required error, the prediction is considered as a hit.

$$a = predictedValue / realValue \qquad (1)$$
$$hit = a \cdot 100 < 100 - error \parallel a \cdot 100 > 100 + error$$

4.1.2 Predictor performance:

More complicated is to define the global performance of a predictor with a given set of executions. Our definition of global performance is based on the percentage of times the different applications are well predicted. This analysis is carried out for each of the predicted values, such as memory usage or total time, because a given predictor may predict better some them that the other.

When evaluating a prediction variable of a given predictor, we define a vector that has 11 intervals. The first interval will contains the percentage of applications that are well predicted 0% of the times, the second one the percentage of applications that are well predicted from 1% to 10% of the times, the third one the number of those that are

well predicted from the 11% to 20% and so on. Those predictors that are carrying out better predictors will have higher numbers on the right side of the vector. For instance we could have a predictor that for a given variable in the interval 81-90% has a value of 50%, what would mean that from the 100% of the predicted applications the 50% of them are well predicted from 81% to 91% of the times.

We decided to create another interval that ponders each interval of the vector explained in the last paragraph for the amount of the predicted *variable* consumed by each application that belongs to the interval in all of its executions. For example, in the case of the system time, we ponder each interval for the amount of the time consumed by the applications that belongs to it. With this second interval we could realize how important the predicted applications were. For example we could see that 1% of applications are well predicted 5% of times, and they represent the 45% of the total time consumed by all the applications.

4.2 Predictors

For this work we have developed seven different predictors based on similar submission behaviours. We support that this conclusion will be also applicable to the Grid environment, because in general, the users that are submitting jobs to our centres are the same users that will submit jobs to our Grids.

- 4.2.1: Last value job: This predictor returns, for a given user and a given application, the amount of the queried resource prediction that the application executed by the same user used in its last execution. With this predictor we expected to predict the applications are not executed with frequency, because other predictors, that require more historical data, would not be reliable.
- 4.2.2 Last value job indexed by tasks: Last value job indexed by tasks: This predictor is pretty similar as the last one, but it also indexes its predictions by the number of tasks used by the execution. A prediction will return the last amount of the queried resource that used the last execution for the given application, the given user and with the given number of tasks. We expected to achieve better prediction with those parallel applications where the amount of the used resource depends on the number of tasks used in the execution.
- 4.2.3 Mean job tasks: This predictor returns the mean for the queried resources of all the previous executions that used the given number of tasks of the given application and for the given user. We expected to hit applications that are having some variability on the amount of used resources in their executions, and that with the last value we miss the predictions due to this variability is higher than a 5%, or the used percentage of error, but in mean this variability is less than the error.
- 4.2.4 Median job tasks: This predictor returns the median for the queried resources of all the previous executions that used the given number of tasks of the given application for the given user. The goal of this predictor is the same as the last one: catch those applications that are having some variability in the amount of used resources in their executions for a given user. However, as the mean is

heavily influenced for the outliers, we wanted to use the non biased estimator median.

– 4.2.5 Using mean and standard deviation: In [11] a formula (2) for memory usage prediction is presented. This formula uses the memory usage of all the past executions for the given user and application.

$$d = executions(application, user)$$
$$prediction = \min(\max(d), mean(d) + 3 \cdot stdev(d)) \quad (2)$$

– 4.2.6 Using median and the IQR : The formula explained on the last predictor was interesting, however as explained in the Mean predictor, the mean and the standard deviation are influenced by the outliers or extreme values, so we decided to implement the same formula but using non biased estimators (3), in order to check how the predictions could be affected by these values.

$$d = executions(application, user)$$
$$prediction = \min(\max(d), median(d) + 3 \cdot IRQ(d)) \quad (3)$$

4.3 Predictors performance

This section presents a general evaluation of the presented predictor performance focused on memory usage, total time and user time for the applications, users and groups contained in the workload.

The workloads were obtained from a Load Leveler three years history files obtained from an IBM SP2 System with two different configurations: the IBM RS-6000 SP with 8*16 Nighthawk Power3 @375Mhz with 64 Gb RAM and the IBM p630 9*4 p630 Power4 @1Ghz with 18 Gb RAM. A total of 336Gflops and 1.8TB of Hard Disk are available. The operating system that they are running is an AIX 5.1

4.3.1 Memory usage prediction in parallel applications

All the predictors had similar behaviour: they forecasting good predictions in 35% of the applications, they had poor prediction results in 30% of the cases, and the 35% left has been spread among the other intervals.

When pondering each application by its number of executions performance of all predictors is dropping substantially, what means that predictors did not do good memory prediction with applications that where executed some more times. Predictors last value, last value indexed by task and median respect the others are achieving best performance. Worst results of predictor mean and predictor presented in 3.2.5 (the

once that uses mean and the standard deviation) could be explained by possible outliers or extreme values that are affecting the biased estimators used in their predictions: mean and standard deviation. That implies that a future work should include analysis of outliers or values that are having a big impact on predictors based on biased estimators, perhaps pondering this extreme values by a factor of ∂ that decrease it weight in the overall prediction would be a way to decrease the effect of such values.

Finally, when taking into account the amount of memory usage of each application, the last value indexed by task respect the others is the once that is carrying out better predictions with such applications that are using more memory.

4.3.2 Total time prediction in parallel applications

Differences between predictors are less than 5 % in each interval, what mean that they are having similar performance in terms of how well are predicting each application.

Pondering each application for number of times that it was executed the last value indexed by task and last value predictors are having better performance with those applications that are more executed. We think that this fact could be caused because total time has some sort of linear relation with the number of used tasks.

Applications that spent more time to be executed are correctly predicted from 60% up to 89% of they total executions by three predictors: last value, last value indexed by tasks and mean. Predictors are doing good predictions with applications that are not executed many times (those that were executed more times were on the interval 30% to 70% for the same predictors). As a conclusion we can say that in general predictions can be improved substantially, but we have detected that an important subset of applications, those are executed few times, are good predicted for some of the predictors, that implies that we could use this kind of predictors for those predict those applications of which we have less historical data.

4.3.3 Memory usage prediction in sequential applications

Prediction performance for all the presented predictors is pretty similar when taking into account the percentage of times that a given application was well predicted. They spear their predictions in two main groups of applications: 40% of applications are almost never well predicted, and another 40% is well predicted from 90% to 100% of times.

When pondering each application for its number of executions those applications that executed more times are well predicted from 1%-10% of times in the mean predictor. However median and last value predictors are achieving best results, this could mean that applications that are being executed more times have an amount of memory usage that are possible outliers or extreme values and are affecting the mean. Pondering applications for its total amount of memory used in general predictors had a poor performance for those applications that used more amount of memory.

4.3.3 Total time prediction in sequential applications

As in the previous analysis, predictors had similar performance in terms of how well they predict the total time for the sequential applications. There are approximately a 50% of applications that are whose executions are well predicted around 90%-100% of times, and there are a 40% of applications that are almost never well predicted. However the importance of the applications that are well predicted is doubtful, due to those applications that are more executed or that are consuming more time are well predicted at most the 30% times of they executions.

5 Conclusions

One of the main problems regarding performance issues in Grid computing is the particular environment where Grid Services and Resources interact among them. We presents and justify a different strategy that intend to self learn from current and historical usability information of Grid Resources and Services in order to determine performance predictions.

For that reason, two early objectives have been defined: to understand the behaviours and characteristics of submitted jobs and to design performance prediction techniques.

The first goal was achieved through the CEPBA-UPC centre workload analysis [12]. Something that has to be taken into account this study was not intended to provide an accurate description of the workload neither to provide specific workload models as can be found in other works[13], we characterized some issues that we considered useful for our purposes and we extracted and analyzed them from our system.

The second one has been completed by designing and evaluating simple prediction techniques. Before developing predictors based more complex and sophisticated techniques, such as those that can be found on [14], we decided to test how well predictors based on simple ideas or algorithms performed on our system.

We have realized that working with real workloads is a tedious and difficult task due to many factors has to be taken into account, and not always is possible achieve clear conclusions. Next phases will include study of synthetic workloads. This will help us to focus the prediction on set of different kind of applications, and we expected that will allow have more precise conclusions.

We have also concluded that there are no predictors that do good predictions for all the applications. It's necessary to design hybrid predictors or specific predictors for a given set of applications. We will need to characterize applications and find out which of their characteristics make them more suitable to be predicted by a specific predictor.

References

1. Poulsen, D.K.; Yew, P.-C, Execution-driven tools for parallel simulation of parallel architectures and applications, Supercomputing '93. Proceedings, 15-19 Nov. 1993 Page(s):860 - 869
2. Malloy, B.A.;Trace-driven and program-driven simulation: a comparison. Modelling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994., MASCOTS '94., 31 Jan.-2 Feb. 1994 Page(s):395 - 396
3. Embra: Fast and Flexible Machine Simulation, Emmett Witchel, Mendel Rosenblum, Massachusetts Institute of Technology and Stanford University. Sigmetrics 1996
4. PERFORM - A Fast Simulator For Estimating Program Execution Time, Alistair Dunlop and Tony Hey, Department Electronics and Computer Science, University of Southampton
5. Candice Bechem, et al; An Integrated Functional Performance Simulator, Carnegie Mellon University, 0272-1732/99/ 1999 IEEE
6. Brett H. Meyer et al; Power-Performance Simulation and Design Strategies for Single-Chip Heterogeneous Multiprocessors. IEEE Transactions On Computers, Vol. 54, No. 6, JUNE 2005
7. Connie U. Smith; Performance Engineering of Software Systems, Reading, MA, Addison-Wesley, 1990.
8. Connie U. Smith and Lloyd G. Williams; Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives, IEEE Transactions on Software Engineering, Vol. 19, No. 7, July 1993
9. Stephen A. Jarvis, Daniel P. Spooner, Helene N. Lim Choi Keung, Graham R. Nudd Performance Prediction and its use in Parallel and Distributed Computing Systems.. High Performance Systems Group, University of Warwick, Coventry, UK
10. D. J. Kerbyson, J. S. Harper, A. Craig, and G. R. Nudd.. PACE: A Toolset to Investigate and Predict Performance in Parallel Systems. In European Parallel Tools Meeting, ONERA, Paris, October 1996. Keyword(s): PACE, Performance Prediction.
11. Anat Batat and Dror G. Feitelson, ``Gang Scheduling with Memory Considerations". In 14th Intl. Parallel & Distributed Processing Symp. (IPDPS), pp. 109-114, May 2000.
12. Francesc Guim Bernat, Julita Corbalan, Jesus Labarta; Analyzing LoadLeveler historical information for performance prediction. XXI Jornadas de Paralelismo. CEDI 2005
13. Evgenia Smirni and Daniel A. Reed, ``Workload Characterization of Input/Output Intensive Parallel Applications". In 9th Intl. Conf. Comput. Performance Evaluation, Springer-Verlag, Lect. Notes Comput. Sci. vol. 1245, pp. 169-180, Jun 1997
14. Warren Smith, Ian Foster, and Valerie Taylor, ``Predicting Application Run Times Using Historical Information". In Job Scheduling Strategies for Parallel Processing, Dror G. Feitelson and Larry Rudolph, (ed.), Springer Verlag, Lect. Notes Comput. Sci. vol. 1459, pp. 122-142, 1998.

Integrating Resource and Service Discovery in the CoreGrid Information Cache Mediator Component

Giovanni Aloisio[1], Zoltán Balaton[2], Peter Boon[3], Massimo Cafaro[1], Italo Epicoco[1], Gábor Gombás[2], Péter Kacsuk[2], Thilo Kielmann[3], and Daniele Lezzi[1]

[1] Center for Advanced Computational Technologies ISUFI,
University of Lecce and
National Nanotechnology Lab/INFM&CNR,
Lecce, Italy
{giovanni.aloisio, massimo.cafaro,
italo.epicoco, daniele.lezzi}@unile.it
[2] MTA SZTAKI
Computer and Automation Research Institute
Hungarian Academy of Sciences, Hungary
{balaton, gombasg, kacsuk}@sztaki.hu
[3] Vrije Universiteit Amsterdam, The Netherlands
{pboon, kielmann}@cs.vu.nl

Abstract. In this paper we describe how the CoreGrid application-level information cache mediator component will benefit from the integration of resource and service discovery mechanisms available in iGrid and Mercury. The former is a novel Grid Information Service based on the relational model. iGrid has been initially developed within the GridLab project by the ISUFI Center for Advanced Computational Technologies (CACT) at the University of Lecce, Italy. It provides fast and secure access to both static and dynamic information through a Globus Toolkit GSI (Grid Security Infrastructure) enabled web service. Besides publishing system information, iGrid also allow publication of user's or service supplied information. The adoption of the relational model provides a flexible model for data, and the hierarchical distributed architecture provides scalability and fault tolerance. The latter, which has also been initially developed within the GridLab project by MTA SZTAKI, has been designed to satisfy requirements of grid performance monitoring: it provides monitoring data represented as metrics via both pull and push access semantics and also supports steering by controls. It supports monitoring of grid entities such as resources and applications in a generic, extensible and scalable way. It is implemented in a modular way with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system.

1 Introduction

The CoreGrid partners of task 7.2 are actively developing a suite of components that mediate between applications and system software [1]. These Mediator Components will be derived from the ongoing developments of the partner institutions involved in this task. The research is focused on the definition and inplementation of a novel grid component architecture; the aim is to foster integration and linking of different grid components with minimal effort, by providing a simple, well defined glue layer between all kind of components. The envisioned architecture also includes a tools framework, consisting of an integrated components Grid platform, a component support toolkit, and a generic problem-solving toolkit, as in Figure 1.

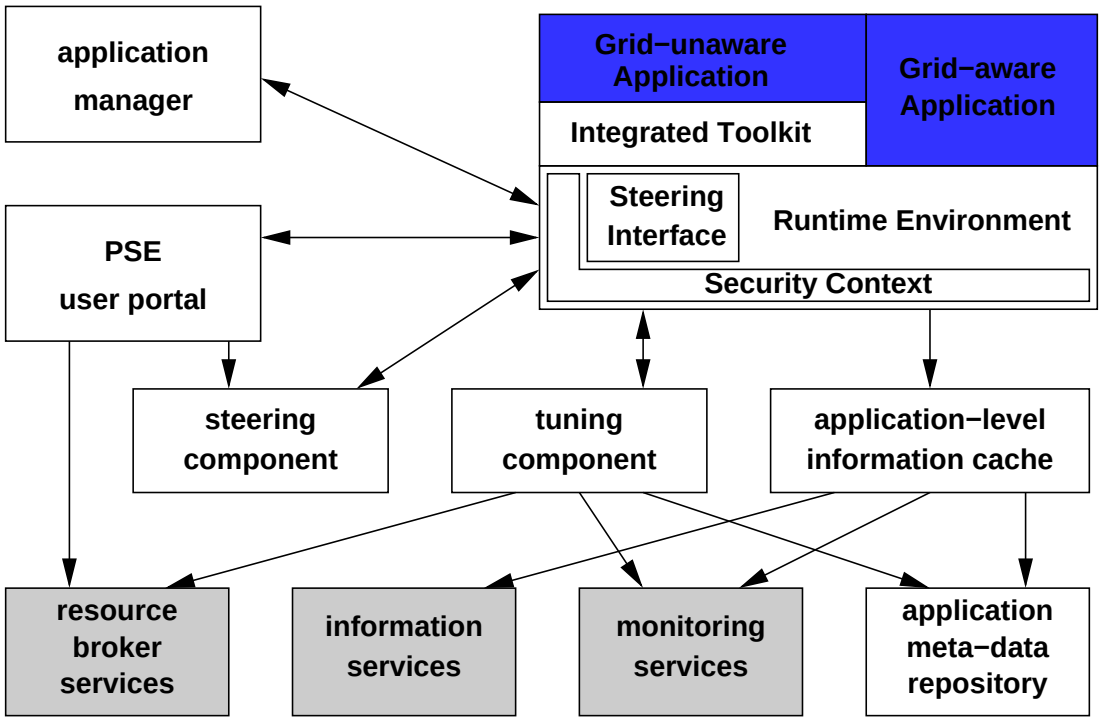


Fig. 1. Grid component architecture

The proposal for a mediator component toolkit includes an application-level information cache mediator component. This component will be designed to provide a uniform interface to access several kinds of different data originating from information services, monitoring services and application-level meta-data. Moreover, a caching mechanism will allow delivering the information to applications and/or components that need it really fast.

We are now jointly collaborating to develop an application-level information cache mediator component. Our activities also include the integration of the iGrid information service [2] [3] and the Mercury monitoring service [4], to provide the envisioned grid component architecture with resource and service discovery capabilities. Indeed, grid environments require the availability of an

information rich environment to support resource and service discovery, and thus decision making processes related to dynamic adaptation. Distributed computational resources and services are sources and/or potential sinks of information; the data produced can be static or dynamic in nature, or even dynamic to some extent. Depending on the actual degree of dynamism, information is better handled by a Grid Information Service (static or quasi-static information) or by a Monitoring Service (highly dynamic information).

In this context, information plays a key role, therefore Grid Information and Monitoring Services are fundamental building block of a grid infrastructure/middleware. Achieving high performance execution in grid environments is virtually impossible without timely access to accurate and up-to-date information related to distributed resources and services: the lack of information about the execution environment prevents design and implementation of so called grid-aware applications. Indeed, an application can not react to changes in its environment if these changes are not advertised. Therefore, self-adjusting, adaptive applications are natural consumers of information produced in grid environments. However, making relevant information available on-demand to consumer applications is nontrivial, since information can be (i) diverse in scope, (ii) dynamic and (iii) distributed across one or more Virtual Organizations. It is worth noting here that obtaining information about the structure and state of grid resources, services, networks etc. can also be challenging in large scale grid environments.

The rest of the paper is organized as follows. We discuss resource and service discovery mechanisms available in iGrid in Section 2, and present the Mercury monitoring service in Section3. We give an overview of the application-level information cache in Section 4, and conclude the paper in Section 5.

2 iGrid

iGrid is a novel Grid Information Service initially developed within the European GridLab project [5] by the ISUFI Center for Advanced Computational Technologies (CACT) at the University of Lecce, Italy. An overview of the iGrid Information Service can be found in [2]; here we delve into details related specifically to resource and service discovery mechanisms available.

iGrid distributed architecture is based on iServe and iStore GSI [6] enabled web services. An iServe collects information related to the computational resource it is installed on, while iStore gathers information coming from trusted, registered iServes. The current architecture resembles the one adopted by the Globus Toolkit MDS, therefore iStores are allowed to register themselves to other iStores, creating arbitrarily complex distributed hierarchies. Even though this architecture proved to be effective to build scalable distributed collections of servers, neverthless we are already investigating peer-to-peer overlay networks based on current state of the art distributed hash table algorithms in order to improve iGrid scalability. The implementation includes system information providers outputting XML, while trusted users and/or services can publish information simply calling a web service registration method. Resource discovery

using the iGrid Information Service is based on the availability of the following information (not exaustive):

System operating system, release version, machine architecture etc;

CPU for CPUs, static information such as model, vendor, version, clock speed is extracted; the system also provides dynamic information such as idle time, nice time, user time, system time and load;

Memory static information such as RAM amount and swap space is available. Dynamic information related to available memory and swap space is published too;

File Systems static as well dynamic information is extracted, such as file system type, mount point, access rights, size and available space;

Network Interfaces network interface names, network addresses and network masks;

Local Resource Manager the information belonging to this category can be further classified as belonging to three different subclasses: information about queues, jobs and static information about Local resource Management System (LRMS). Some examples of extracted information are: LRMS type and name; queue name and status, number of CPU assigned to the queue, maximum number of jobs that can be queued, number of queued jobs, etc; job name, identifier, owner, status, submission time etc. Currently information providers for OpenPBS and Globus Gatekepeer are available, with LSF planned;

Certification Authorities certificate subject name, serial number, expiration date, issuer, public key algorithm etc.

Virtual Organization information related to VO can be used to automatically discover which resources belong to a given VO; we have VO name, resource type, help desk phone number, help desk URL, job manager, etc.

Of course, this set of information is not meant to be static, the iGrid schema will continue to evolve and will be extended to support additional information as required by the GridLab project or iGrid users.

One of the most important requirements for grid computing scenarios is the ability to discover services and web/grid services dynamically. Services in this context refers to traditional unix servers. The iGrid system provides users and developers with the following functionalities: register, unregister, update and lookup. More than one instance for each service or web service can be registered. The following information is available for services: logical name, instance name, service description, default port, access URL, distinguished name of the service publisher, timestamps related to date of creation an date of expiration of the published information.

For web services, relevant information includes logical name, web service description, WSDL location (URL), web service access URL, distinguished name of publisher and timestamps related to date of creation and date of expiration of the published information.

Information related to firewalls is strictly related to service information. As a matter of fact, before registering a service, developers will query iGrid to retrieve

the range of open ports available on a specified computational resource. This is required in order to chose an open port, allowing other people/services to connect to a registered service. The information available includes firewall hostname, open ports, time frame during which each port (or a range of ports) is open, the protocol (TCP/UDP) used to connect to these ports, the distinguished name of the firewall administrator, and timestamps related to date of creation and date of expiration of the published information.

iGrid uses a push model for data exchange. Indeed, system information (useful for resource discovery) extracted from resources is stored on the local database, and periodically sent to registered iStores, while user and/or service supplied information (useful for service discovery) is stored on the local database and immediately sent to registered iStores. Thus, an iStore has always fresh, updated information related to services, and almost fresh information related to resources; it does not need to ask iServes for information. The frequency of system information forwarding is based on the information itself, but we also allow defining a per information specific policy. Currently, system information forwarding is based on the rate of change of the information itself. As an example, information that does not change frequently or change slowly (e.g. the amount of RAM installed) does not require a narrow update interval. Interestingly, this is true even for the opposite extreme, i.e., for information changing rapidly (e.g., CPU load), since it is extremely unlikely that continuous forwarding of this kind of information can be valuable for users, due to information becoming quickly inaccurate. Finally, information whose rate of change is moderate is forwarded using narrow update intervals.

We have found that the push model works much better than the corresponding pull model (adopted, for instance, by the Globus Toolkit MDS) in grid environments. This is due to the small network traffic volume generated from iServe to iStore servers: on average, no more than one kilobyte of data must be sent. Moreover, we tag information with a time to live attribute that allows iGrid to safely removes stale information from the database when needed. For instance, when users search for data, a clean-up operation is performed before returning to the client the requested information, and during iGrid system startup, the entire database is cleaned up. Therefore the user will never see stale information.

Finally, it is worth recalling here that the performances of iGrid are extremely good, as reported in [2] .

3 Mercury

In a complex system as the grid, monitoring is essential for understanding its operation, debugging, failure detection and for performance optimisation. To achieve this, data about the grid must be gathered and processed to reveal important information. Then, according to the results, the system may need to be controlled. The Mercury Grid Monitoring System provides a general and extensible grid monitoring infrastructure. Mercury Monitor is designed to satisfy specific requirements of grid performance monitoring [7]. It provides monitoring

data represented as metrics via both pull and push model data access semantics and also supports steering by controls. It supports monitoring of grid entities such as resources and applications in a generic, extensible and scalable way. The architecture of Mercury Monitor extends the Grid Monitoring Architecture (GMA) [8] proposed by Global Grid Forum with actuators and controls. Mercury Monitor features a modular implementation with emphasis on simplicity, efficiency, portability and low intrusiveness on the monitored system.

The input of the monitoring system consists of measurements generated by sensors. Sensors are controlled by producers that can transfer measurements to consumers when requested, and are implemented as shared objects that are dynamically loaded into the producer at run-time depending on configuration and incoming requests for different measurements. It is also important to note that in Mercury measurements are performed only when requested by a consumer and data is only sent where it is needed. All measurable quantities are represented as metrics. Metrics are defined by a unique name such as *host.cpu.user* which identifies the metric definition, a list of formal parameters and a data type. By providing actual values for the formal parameters a metric instance can be created, representing a specific entity to be monitored. A measurement corresponding to a metric instance is called a metric value. Values contain a timestamp and the measured data according to the data type of the metric definition. Sensor modules implement the measurement of one or more metrics. Mercury Monitor supports both event-like (i.e. an external event is needed to produce a metric value) and continuous metrics (i.e. a measurement is possible whenever a consumer requests it, e.g., the CPU temperature in a host). Continuous metrics can be made event-like by requesting automatic periodic measurements.

The GMA proposal of the Global Grid Forum only describes components required for monitoring. It is often necessary however, to also influence the monitored entity based on the analysis of measured data. For example, an application might need to be told to checkpoint and migrate if it does not perform as expected, a service may need to be restarted if it crashed or system parameters (such as process priorities or TCP buffer sizes) might need to be adjusted depending on current resource usage. To support this, actuators have been introduced in Mercury. Actuators are analogous to sensors in the GMA but instead of monitoring something they provide a way to influence the monitored entity. As sensors are accessed by consumers via producers, actuators are made available for consumers via actuator controllers. As the producer manages sensors (start, stop and control sensors and initiate measurements on a user's request) the actuator controller manages actuators. Similarly to metrics implemented by sensors, actuators implement controls that represent interactions with either the monitored entities or the monitoring system itself. The functional difference between metrics and controls is that metrics only provide data while controls do not provide data except for a status report but they influence the state or behaviour of the monitoring system or the monitored entity.

Besides providing information about grid resources, Mercury contains two elements to aid application and service monitoring and steering: an application

sensor and an instrumentation library that communicates with this sensor. Together they allow to register application specific metrics and controls and to receive and serve requests for metrics and controls while the application is running. The application sensor is resposible for keeping track of processes of jobs as well as any private metrics or controls that the running applications provide. The application sensor forwards requests for application specific metrics or controls to the application process(es) they belong to. The request is then interpreted by the instrumentation library by performing the measurement or executing the requested control. The instrumentation library communicates with the application sensor using a UNIX domain socket, but it also has shared memory support to speed up transferring large volumes of data such as generated by fine-grained instrumentation. The application may also put certain variables into the shared memory area so they can be queried or modified without direct interaction with the application. This is useful for single-threaded applications or services that do not call the event handler of the instrumentation library for extended periods of time. Processing of application specific metric events is optimised inside the instrumentation library, i.e. they will not be sent to the application sensor if there are no consumers requesting them. This ensures that if an application is built with fine-grained instrumentation it can still run without noticeable performance degradation if the generated events are not requested.

4 Application-level Information Cache

The envisioned application-level information cache component [1] is supposed to provide a unified interface to deliver all kinds of meta-data (e.g., from a GIS like iGrid, a monitoring system like Mercury, or from application-level meta data) to a grid application. The cache's purpose is twofold. First, it is supposed to provide a unifying component interface to all data (independent of its actual storage), including mechanisms for service and information discovery. Second, this application-level cache is supposed to deliver the information really fast, cutting down access times of current Web-service based implementations like Globus GIS (up to multiple seconds) to the order of a method invocation. For the latter purpose, this component may have to prefetch (poll) information from the various sources to provide them to the application in time.

Such an application-cache component is currently being developed as a collaboration among the authors. Its API as presented to the application is inspired by GridLab's GAT [9]. The GAT specifies an API for monitoring purposes. The basis of this API is general and extensible enough to fit the current monitoring and information systems and possibly future ones too. Like Mercury and GAT, the mediator defines measurable quantaties as metrics. A metric is a container which holds the unique name of the metric and the properties needed to retrieve the information, like parameters. The result of a measurement is stored in a container called a metric value.

An application that requests information creates a metric which specifies the source host from which the information originates, the possibly required param-

eters for the metric, and a recommended frequency which indicates how often the mediator should update the corresponding metric value in its cache. Note that the frequency could be omitted if the underlying monitoring or information system is able to push the requested information.

From the moment the mediator receives the first metric value, the application will be able to get it from the cache without any delays. However, it could also choose to get notified if a value gets updated, this way both pull and push mechanisms are available to the application. Another option for applications is to request a metric value bypassing the cache. In this case, the mediator component will merely serve as a uniform interface between the underlying monitoring and information systems and the application.

In order to serve information from different sources, the mediator component uses an extensible 'plugin' system, where each plugin (metric provider) forms the link between the mediator component and the underlying monitoring or information system. The metric providers have to deal with the actual retrieval of information and present it to the (rest of the) mediator component. The mediator component will process the information further, possibly by caching it.

A metric provider instance will represent one running information system on a host. So multiple metric provider instances retrieving information from the same type of information system, only from different hosts can exist next to each other. Therefore, when an application does not specify a metric provider to use when retrieving a metric, it should at least specify the source host running the information system. That way, the mediator component is able to group metric providers retrieving information from the same host.

Some types of information may be retrieved from only one information system, while other types of information could be obtained from multiple. This latter type of information could be presented by the different systems in different ways, using different data types or even different measurement units. It is up to the metric providers to translate information presented by the underlying system into a format which the mediator component presents to the application.

If information is requested by an application, the application can either choose a certain metric provider, or leave it up to the mediator to decide which metric provider will be used to retrieve the information. If multiple metric providers are able to retrieve the same type of information, it is a matter of policy which one is chosen (lowest response time, most reliable).

The currently developed prototype is providing the following interface:

List getMetricDefinitions()
Gets a list of available MetricDefinitions.

MetricDefinition getMetricDefinitionByName(String name)
Gets a MetricDefinition given the metric name.

MetricValue getMetricValue(Metric metric)
Retrieves a MetricValue from the cache, if available. Throws an exception if the value is not stored in the cache.

void startProviding(Metric metric)
Tells the cache to retrieve the given metric. The cache tries to keep the metric

value updated according to the frequency set in the metric parameter. No guarantees can be given, however, whether the (underlying) system is capable of providing the information at this rate.

void stopProviding(Metric metric)
Tells the cache that the metric described by the parameter does not need to be updated anymore.

void addMetricListener(MetricListener listener, Metric metric)
Adds a MetricListener. The listener object will be called whenever an updated value becomes available for the metric.

void removeMetricListener(MetricListener listener, Metric metric)
Removes a MetricListener. No notification through the listener will be sent any more.

MetricProviderManager getMetricProviderManager()
Returns the MetricProviderManager for advanced configuration.

MetricValue getMetricValueFromProvider(Metric metric)
Retrieves a MetricValue directly from a MetricProvider, bypassing the cache. The MetricProvider should be specified in the Metric object.

MetricListener (Interface)
When a metric value is updated in the cache, applications can retrieve events through objects implementing the MetricListener interface.

void processMetricEvent(MetricValue val)
An instance of a class implementing this interface receives MetricValues through calls to this method when it is registered to receive such events.

5 Conclusion

We have described the iGrid information service and the Mercury monitoring service, focusing our attention to the integration of these and other sources of useful information in an application-level information cache mediator component we are jointly developing in the context of the European CoreGrid project. We gave an overview of the forthcoming mediator component, including details related to its API. This component will provide a basis for dynamic adaptation in grid environment, as envisioned in the CoreGrid grid component architecture. As its implementation is ongoing work, quantitative evaluations are not available yet.

Acknowledgements

This work is partially funded by the European Commission, via the Network of Excellence *CoreGRID* (contract 004265).

References

1. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems: Proposal for mediator component toolkit. CoreGRID deliverable D.ETS.02 (2005)

2. Aloisio, G., Cafaro, M., Epicoco, I., Fiore, S., Lezzi, D., Mirto, M., Mocavero, S.: igrid, a novel grid information service. In: Proceedings of Advances in Grid Computing - EGC 2005. Volume 3470., Lecture Notes in Computer Science, Springer-Verlag (2005) 506–515
3. Aloisio, G., Cafaro, M., Epicoco, I., Fiore, S., Lezzi, D., Mirto, M., Mocavero, S.: Resource and service discovery in the igrid information service. In: Proceedings of International Conference on Computational Science and its Applications (ICCSA 2005). Volume 3482., Springer-Verlag (2005) 1–9
4. Gombás, G., Balaton, Z.: A flexible multi-level grid monitoring architecture. In: Proceedings of the First European Across Grids Conference. (2003)
5. Allen, G., Davis, K., Dolkas, K.N., Doulamis, N.D., Goodale, T., Kielmann, T., Merzky, A., Nabrzyski, J., Pukacki, J., Radke, T., Russell, M., Seidel, E., Shalf, J., Taylor, I.: Enabling Applications on the Grid – A GridLab Overview. International Journal on High Performance Computing Applications **17**(4) (2003) 449–466
6. Foster, I., Kesselmann, C., Tsudik, G., Tuecke, S.: A security architecture for computational grids. In: Proceedings of 5th ACM Conference on Computer and Communications Security Conference. (1998) 83–92
7. Németh, Z., Gombás, G., Balaton, Z.: Performance evaluation on grids: Directions, issues, and open problems. In: Proceedings of the Euromicro PDP 2004, A Coruna, Spain, IEEE Computer Society Press (2004)
8. Fisher et al., S.: R-gma: A relational grid information and monitoring system. In: Proceedings of 2nd Cracow Grid Workshop, Cracow, Poland (2003)
9. Allen, G., Davis, K., Goodale, T., Hutanu, A., Kaiser, H., Kielmann, T., Merzky, A., van Nieuwpoort, R., Reinefeld, A., Schintke, F., Schütt, T., Seidel, E., Ullmer, B.: The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. Proceedings of the IEEE **93**(8) (2005) 534–550

Redesigning the SEGL Problem Solving Environment: A Case Study of Using Mediator Components

Thilo Kielmann[1], Gosia Wrzesinska[1], Natalia Currle-Linde[2], and Michael Resch[2]

[1] Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{kielmann|gosia}@cs.vu.nl
[2] High Performance Computing Center (HLRS), University of Stuttgart, Germany
{linde|resch}@hlrs.de

Abstract. The Science Experimental Grid Laboratory (SEGL) problem solving environment allows users to describe and execute complex parameter study workflows in Grid environments. Its current, monolithic implementation provides much high-level functionality for executing complex parameter-study workflows. Alternatively, using a toolkit of mediator components that integrate system-component capabilities into application code would allow to build a system like SEGL from existing, more generally applicable components, simplifying its implementation and maintenance. In this paper, we present the given design of the SEGL PSE, analyze the provided functionality, and identify a set of mediator components that can generalize the functionality required by this challenging application category.

1 Introduction

The SEGL problem solving environment [5] allows end-user programming of complex, computation-intensive simulation and modeling *experiments* for science and engineering. Experiments are complex workflows, consisting of domain-specific or general purpose simulation codes, referred to as *tasks*. For each experiment, the tasks are invoked with input parameters, that are varied over given parameter spaces, together describing individual parameter studies. SEGL allows users to program so-called *applications* using a graphical user interface. An application consists of several tasks, the *control flow* of their invocation, and the *data flow* of input parameters and results. For the parameters, the user can describe iterations for parameter sweeps; also, conditional dependencies on result values can be part of the control flow. Using such a user application program, SEGL can execute the tasks, provide them with their respective input parameters, and collect the individual results in an experiment-specific database.

In this paper, we revisit our view of component-based Grid application environments, present the given architecture of the SEGL PSE and its functionality, and identify a set of mediator components that can generalize the functionality

required by this challenging application category. An important insight is the requirement of a persistent application-execution service.

2 Component-based Grid application environments

A technological vision is to build Grid software such that applications and middleware will be united to a single system of components [3]. This can be accomplished by designing a toolkit of components that mediate between both applications and system components. The goal is to integrate system-component capabilities into application code, achieving both steering of the application and performance adaptation by the application to achieve the most efficient execution on the available resources offered by the Grid.

By introducing such a set of components, resources and services in the Grid get integrated into one overall system with homogeneous component interfaces. The advantage of such a component system is that it abstracts from the many software architectures and technologies used underneath. The strength of such a component-based approach is that it provides a homogeneous set of well-defined (component-level) interfaces to and between all software systems in a Grid platform, ranging from portals and applications, via mediator components to the underlying system software. A possible set of envisioned mediator components can be seen in Figure 1. We elaborate on them in the following.

Runtime Environment The runtime environment implements a set of component interfaces to various kinds of Grid services and resources, like job schedulers, file systems, etc. Doing so, the runtime environment provides an abstraction layer between application components and system components, while explicitly maintaining the application's security context. The interfaces are supposed to be implemented by a delegation mechanism that forwards invocations to service providers. Examples of such runtime environments are the GAT [1], or the platform as envisioned by GGF's SAGA-RG research group.

Steering Interface A dedicated part of the runtime environment is the steering interface. This component-level interface is supposed to make applications accessible by system and user-interface components (like portals, PSE's, or an application manager) like any other component in the system. One obvious additional use of such an interface would be a debugging interface for Grid applications.

Application-level meta-data repository This repository is supposed to store meta data about a specific application, storing, e.g., timing or resource requirements from previous, related runs. The collected information will be used by other components to support resource management (location and selection) and to optimize further runs of the applications automatically.

Application-level information cache
This component is supposed to provide a unified interface to deliver all kinds of meta-data (e.g., from a GIS, a monitoring system, from application-level

meta data) to the application. Its purpose is twofold. First, it is supposed to provide a unifying component interface to all data (independent of its actual storage), including mechanisms for service and information discovery. Second, this application-level cache is supposed to deliver the information really fast, cutting down access times of current implementations like Globus GIS (up to multiple seconds) to the order of a method invocation

Application steering and tuning components Controlling and steering of applications by the user, e.g. via application managers, user portals, and PSE's, requires a component-level interface to give external components access to the application. Besides the steering interface, also dedicated steering components will be necessary, both for mediating between application and system components, but also for implementing pro-active steering systems, carrying their own threads of activity.

Application Manager Component Such a component establishes a pro-active user interface, in charge of tracking an application from submission to successful completion. Such an application manager will be in charge of guaranteeing such successful completion in spite of temporary error conditions or performance limitations.

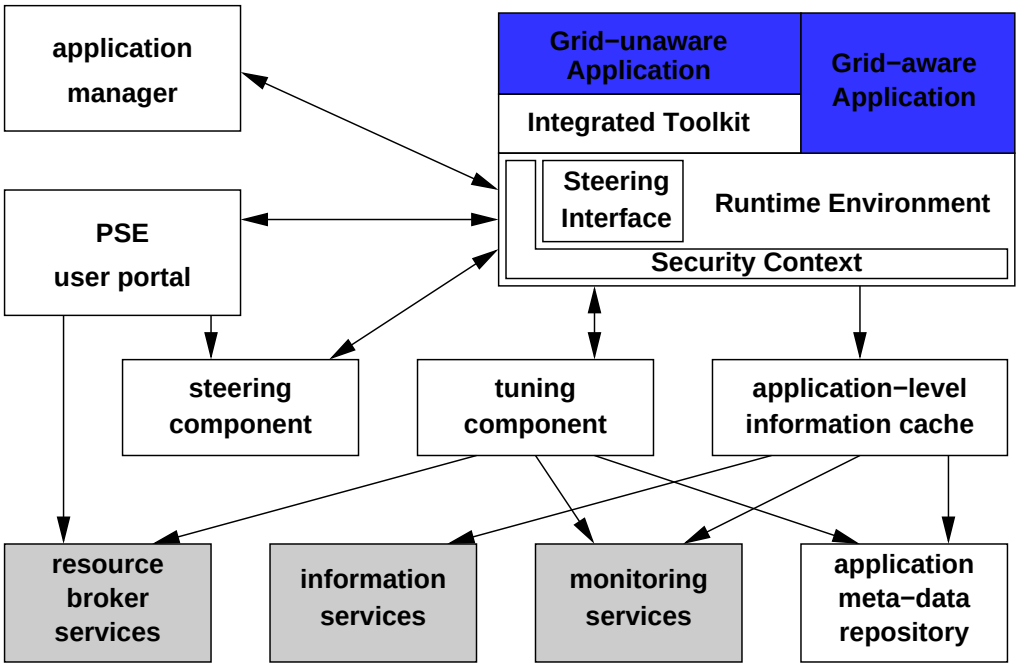


Fig. 1. Envisioned generic component platform

3 The SEGL system architecture

Figure 2 shows the current system architecture of SEGL. It consists of three main components: the User Workstation (Client), the Experiment Application

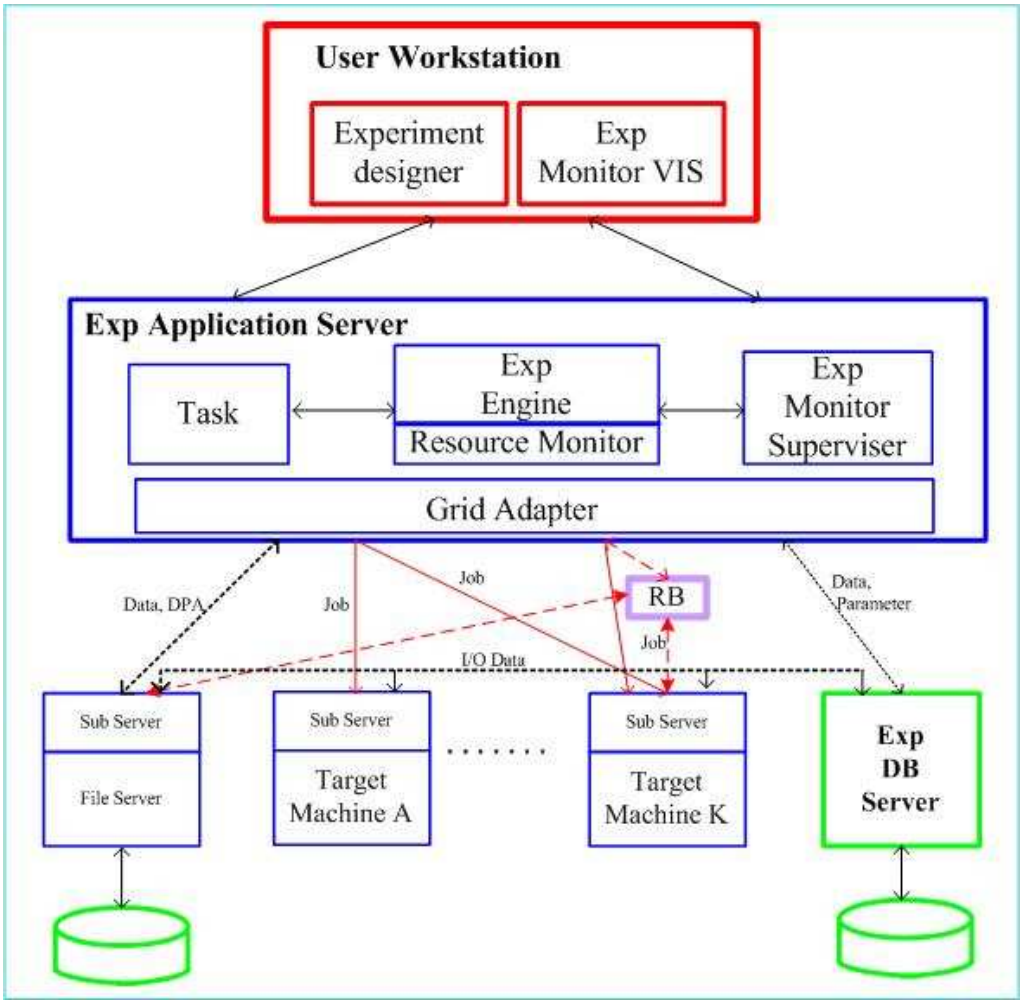


Fig. 2. Current SEGL architecture

Server (ExpApplicationServer), and the Experiment database server (ExpDB-Server). Client and ExpApplicationServer communicate with each other using a traditional client/server architecture, based on J2EE middleware. The interaction between ExpApplicationServer and the Grid resources is done through a Grid Adaptor, interfacing to Globus [6] and UNICORE [8] middleware.

The client on the user's workstation is composed of the graphical experiment designer tool (ExpDesigner) and the experiment process monitoring and visualization tool (ExpMonitorVIS). The ExpDesigner is used to design, verify and generate the experiment's program, organize the data repository and prepare the initial data, using a simple graphical language.

Each experiment is described at three levels: control flow, data flow and the data repository. The control flow level describes which code blocks will be executed in which order, possibly augmented by parameter iterations and conditional branches. Each block can be represented as a simple parameter study. An example is shown in Fig. 3. The data flow level describes the flow of parameter data between the individual code blocks. On the data repository level, a common

description of the metadata repository is created for the given experiment. The repository is an aggregation of data from the blocks at the data flow level.

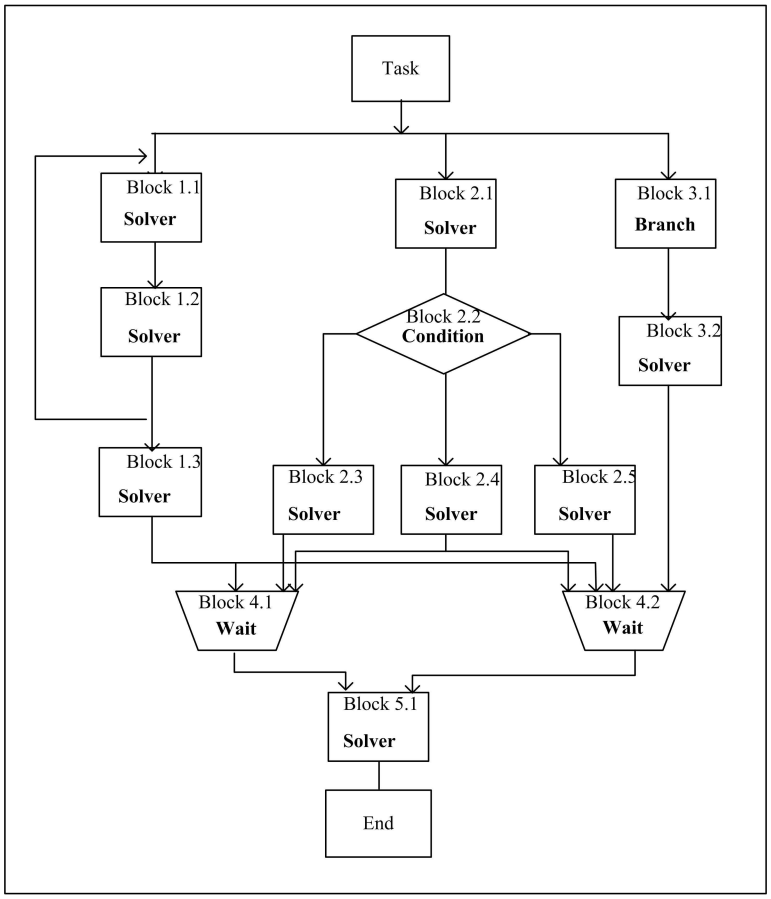


Fig. 3. Example experiment control flow

After completing the graphical design of the experiment program, it is "compiled" to the *container application*. This creates the experiment-specifc parts for the ExpApplicationServer as well as the experiment's data base schema. The container application of the experiment is transferred to the ExpApplicationServer and the schema descriptions are transferred to the server data base. Here, the meta data repository is created.

The ExpApplicationServer consists of the experiment engine (ExpEngine), the container application (Task), the controller component (ExpMonitorSupervisor) and the ResourceMonitor. The ResourceMonitor holds information about the available resources in the Grid environment. The MonitorSupervisor controls the work of the runtime system and informs the Client about the current status of the jobs and the individual processes. The ExpEngine is executing the application Task, so it is responsible for actual data transfers and program executions on and between server machine in the Grid.

The final component of SEGL is the data base server (ExpDBServer). The automatic creation of the experiment is done according to the structure designed

by the user. All data produced during the experiment such as input data for the parameter study, parameterization rules etc are kept in the ExpDBServer.

As SEGL parameter studies may run for significant amounts of time, application progress monitoring becomes necessary. The MonitorSupervisor, being part of the experiment application server, monitors the work of the runtime system and notifies the client about the current status of the jobs and the individual processes. The ExpEngine is the actual controller of the SEGL runtime system. It consists of three sub systems: the TaskManager, the JobManager and the DataManager. The TaskManager is the central dispatcher of the ExpEngine. It coordinates the work of the DataManager and the JobManager as follows:

1. It organizes and controls the execution sequence of the program blocks. It starts the execution of the program blocks according to the task flow and the conditions within the experiment program.
2. It activates a particular block according to the task flow, selects the necessary computer resources for the execution of the program and deactivates the block when this section of the program has been executed.
3. It informs the MonitorSupervisor about the current status of the program.

The DataManager organizes data exchange between the ApplicationServer and the FileServer and between the FileServer and the ExpDBServer. Furthermore, it provides the tasks processes with their the input parameter data. For progress monitoring, the MonitorSupervisor is tracking the status of the ExpEngine and its sub components. It forwards status update events to the ExpMonitorVIS, closing the loop to the user. SEGL's progress monitoring is currently split in to parts:

1. The experiment monitoring and visualization on the client side (ExpMonitor VIS). It is designed for visualizing the execution of the experiment and its computation processes. The ExpMontitorVis allows the user to start, stop, the experiment, and to change the input data and to subsequently re-start the experiment or some part of it.
2. The MonitorSupervisor within the application server controls and observes the work of the runtime system (Exp Engine). It sends continuous messages to the ExpMonitorVis on the client workstation.

This subdivision allows the user to disconnect from its running experiment. In this case, all status update messages will be stored with the application server for delivery to the client as soon as it will become reconnected.

4 Extracting mediator components from the SEGL functionality

The SEGL system constitutes an interesting use case for component-based Grid systems as it comprises all functionality required for complicated task-flow applications. In this section, we try to identify, within the existing SEGL implementation, generic functionality that could be implemented in individual, re-usable or exchangable components.

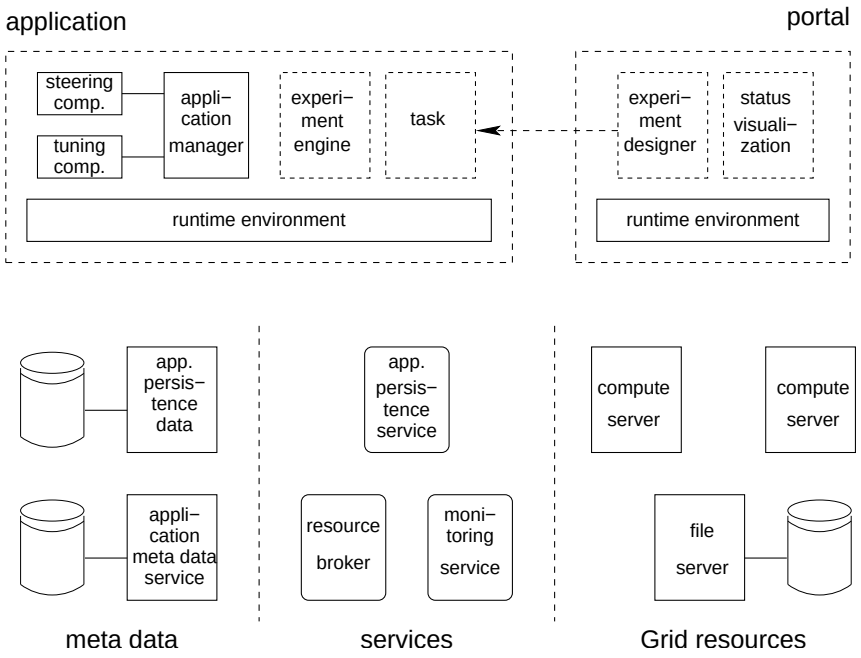


Fig. 4. SEGL redesigned using mediator components

In the current SEGL architecture, as shown in Fig. 2, there is a subdivision to three major areas: the user interface, the experiment application server, and the Grid resources and services. the latter consist of file servers for the experiment's data, compute servers for experiment tasks, and additionally the experiment database, storing all experiment-specific status information. The user interface consists of the experiment programming environment (the "designer") and the application execution visualization component.

By far the most interesting element of SEGL is the experiment application server. It concentrates the application logic (implemented via the *experiment engine* and the experiment-specific *task*), a Grid middleware interface layer (called *adaptor*), as well as progress monitoring functionality. Less visible in Fig. 2 is the fact that the experiment application server is a persistently running service. It has been designed as such to decouple the user interface from possibly long-running experiment codes.

Having such a persistently running service is certainly necessary to guarantee application completion in spite of transient error conditions, without user involvement. However, adding such a domain-specific, permanent service to the pre-installed middleware is causing administrative and security-related concerns.

Based on this analysis, we propose the following re-design based on mediator components, trying to refactor SEGL's functionality into domain-specific components, complemented by general-purpose, reusable components. This redesign is shown in Fig. 4.

In this design, the software Grid infrastructure is organized in three tiers: resources, services, and meta data. For SEGL, relevant Grid resources are both compute and file servers, machines able to execute experimentation tasks and

providing the application data. These servers are accessible via Grid middleware, whichever happens to be installed on each resource.

Relevant Grid services are a resource monitoring service, like e.g. Delphoi [7] and a resource broker that matches tasks to compute servers. For the Grid services, we also propose an *application persistence service*. This is a persistent service, that keeps track of a given application and ensures it runs until successful completion, possibly restarting it in case of failures. Beeing a general-purpose, domain-independent service, it can be deployed in a virtual organization without overly administrative efforts, relying on a security concept that needs to be delpoyed only once for all kinds of applications. In a component-based architecture, we assume these services to have interfaces that fit into the component model.

The final infrastructure category is meta data. For persistent storage of such meta data, one or more servers can be deployed. One such component is the application meta data repository, equivalent to SEGL's current experiment data base. In addition, a meta data storage component is needed for the status information of the application persistence service.

The Grid infrastructure is used by two programs, the SEGL *application* and a user *portal*. Within these programs, Fig. 4 shows general-purpose components as solid boxes and domain-specific components as dashed boxes. Both programs are using the runtime environment for comunication with the Grid infrastructure. The portal is implementing both the experiment designer as well as the experiment status visualization.

SEGL's monitoring and steering facilities also gets split across *application* and *portal*. Within the portal, the status visualization provides the user interface. Within the application, the *steering component* handles change requests for the parameter data. To allow the user to disconnect and later re-connect to his or her application, also the progress monitoring needs storage for its events that is persistent, at least until completion of the overall experiment. For this purpose, the *application meta data service* provides the appropriate storage facilities. The actual progress monitoring then takes place within the *application manager* component, but possibly a dedicated application monitoring and event handling component could be added.

The SEGL application is composed of components only. The experiment engine implements the SEGL-specific application logic, while the task component is created by the experiment designer within the SEGL portal. The experiment engine is accompanied by the generic application manager component which is responsible for both runtime optimization, using dedicated tuning and steering components, and for registering the SEGL application with the application persistence service. In the proposed combination, the experiment engine is responsible for the SEGL-specific control flow, while the application manager is in charge of all Grid-related control aspects, leading to a clear separation of concerns.

5 Conclusions

The SEGL problem solving environment allows end-user programming of complex, computation-intensive simulation and modeling *experiments* for science and engineering. As such, it constitutes an interesting use case for component-based Grid systems as it comprises all functionality required for complicated task-flow applications. In this paper, we have identified, within the existing, monolithic SEGL implementation, generic functionality that can be implemented in individual, reusable components. We have proposed a three-tier Grid middleware architecture, consisting of the resources themselves, persistent services, and meta data. The necessity of a persistent application-execution service was an important insight. Based on this architecture, we were able to compose a SEGL experiment execution application from mostly general-purpose components, augmented only by a SEGL-specific experiment engine and the dynamically created experiment task description. With this architecture we tried to refactor a system like SEGL such that general-purpose functionality is implemented in reusable components while a minimal set of domain-specific components can be added to compose the overall application.

With currently available technology, such components do not exist yet, as suitable component models, and especially generally accepted and standardized interfaces, are subject to ongoing work [4]. Once such components become available [2], refactoring SEGL's implementation will be an interesting excercise.

References

1. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.
2. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems. Proposal for mediator component toolkit. CoreGRID deliverable D.ETS.02, 2005.
3. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems. Roadmap version 1 on Problem Solving Environments, Tools, and GRID Systems. CoreGRID deliverable D.ETS.01, 2005.
4. CoreGRID Virtual Institute on Programming Models. Proposal for a Common Component Model for GRID. CoreGRID deliverable D.PM.02, 2005.
5. N. Currle-Linde, U. Küster, M. Resch, and B. Risio. Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems. In *ParCo 2005*, Malaga, Spain, 2005.
6. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.
7. J. Maassen, R. V. van Nieuwpoort, T. Kielmann, K. Verstoep, and M. den Burger. Middleware Adaptation with the Delphoi Service. *Concurrency and Computation: Practice and Experience*, 2006. Special issue on Adaptive Grid Middleware.
8. D. Erwin (Ed.). *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003.

Integrating Deployment and File Transfer Tools for the Grid

Françoise Baude[2], Denis Caromel[2], Mario Leyton[1], Romain Quilici[2]

[1]DCC Universidad de Chile, and OSCAR**
mleyton@dcc.uchile.cl
[2]OASIS Project, INRIA Sophia-Antipolis, CNRS-I3S, UNSA.
First.Last@sophia.inria.fr

Abstract. Deployment and File Transfer on the Grid corresponds to the struggle between two different interests. On one hand, the goal of achieving an abstract generic approach; on the second one, the objective of supporting a variety of architectures with specific features.
To implement our approach, we have chosen ProActive, since the *ProActive Deployment* compromise corresponds to the *Descriptor Model.* In this model, three different levels of abstractions are defined, from the most abstract to most concrete: VirtualNodes, Java Virtual Machines and Infrastructure (Process). Applying this approach, user programing code can be *liberated* from the burden of handling the architecture specific details.
The present work focus on how to integrate heterogeneous File Transfer tools, specially (but not exclusively), focusing in the *ProActive Descriptor Deployment Model,* while maintaining the balance between the abstract and the concrete representations.

1 Introduction

Transferring files has been a computer science topic, even before the arrival of the Internet, simply by transferring files from one data storage unit to the neighbouring one (unix: *cp*, *mv*). With the introduction of networks, many File Transfer protocols were developed to transfer files from one network node to the next (*rcp, scp, ftp, etc...*).

When dealing with Grid, related middlewares have provided their own tools for deploying jobs and, to some extent, transferring files. From the less sophisticated job schedulers that do not provide file transfer support (LSF, PBS, SGE)[8,9,10], to the frameworks and toolkits that do (GAT, CoG, Unicore, Nordugrid) [1,13,12,7,4].

The wide heterogeneousness of protocols in both dimensions (File Transfer or Deployment) stresses the need of providing an homogeneous integrated mechanism for the Grid. This mechanism should be user flexible, in the sense that

** OSCAR is a joint collaboration project between the *INRIA OASIS Project* and *Universidad de Chile.*

allows the user to combine file transfer protocols with deployment protocols as he/she sees fit. The homogeneousness and flexibility also imply that the mechanism must provide file transfer support for deployment protocols that do not have this support themselves.

In this paper, we describe a proposal for file transfering tools in ProActive[11], specially by focusing on how to integrate third party tools into the *Deployment Descriptor Model*[2]. In the same way that ProActive has managed to integrate different job submission protocols into the *Deployment Descriptor Model*, the proposed File Transfer Model uses a **unique** platform for integrating heterogeneous File Transfer tools.

This document is organized as follows: In section 2 we provide some background on *ProActive Descriptor Deployment Model*. In section 3 we describe our *ProActive File Transfer Model*. In section 4 we show our current implementation and how the model is integrated into the *ProActive Descriptor Deployment Model*. Finally in section 5, we provide some results and benchmarks of our current implementation.

2 ProActive Descriptor Deployment Model Background

Within the *ProActive Descriptor Deployment Model*, it is possible to deploy applications on sites that use heterogeneous protocols, without changing the application source code. All information related with the deployment of the application is described in the XML Deployment Descriptor. Thus, eliminating references inside the code to: machine names, submission protocols (local, rsh, ssh, lsf, globus, unicore, pbs, lsf, etc..) and registry/lookup protocols (rmi, jini, http, etc...).

To achieve this, three levels of abstraction are defined. From the most abstract to the most concrete:

VirtualNodes are abstractions for the location of resources, corresponding to the actual references in the application code. They have a unique identifier, and can be mapped on to one or several Java Virtual Machines (JVM). The result of this mapping corresponds to a set of ProActive Nodes.

JVM stands for the Java Virtual Machines that contain the ProActive Nodes. These JVMs can be created or acquired (on local or remote sites), through the process mapping.

Process corresponds to the mechanism by which JVMs are created or acquired. All the protocol specific information is detailed in the process section.

Effectively, a user can change the mapping of the VirtualNode->JVM->Process to deploy on a different site, without modifying a single line of code in the application. Also note that VirtualNodes are structuring abstractions for capturing the distributed capability of a given application. Typically, several VirtualNodes are used for one application or one Grid component.

Figure 1 shows the deployment configuration of a VirtualNode named *Example* which is mapped into a newly created JVM on a remote machine using the *ssh* protocol.

```
<ProActiveDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="DescriptorSchema.xsd">
 <componentDefinition>
  <virtualNodesDefinition>
   <virtualNode name="Example"/>
  </virtualNodesDefinition>
 <componentDefinition/>
 <deployment>
  <mapping>
   <map virtualNode="Example"><jvmSet><vmName value="Jvm1"/></jvmSet></map>
  </mapping>
  <jvms>
   <jvm name="Jvm1">
    <creation> <processReference refid="sshProcess"/> </creation>
   </jvm>
  </jvms>
 </deployment>
 <infrastructure>
  <processes>
   <processDefinition id="sshProcess">
    <processReference refid="jvmProcess"/>
    <sshProcess class="org.objectweb.proactive.core.process.SSHProcess"
       hostname="example.host" username="smith"/>
   </processDefinition>
   <processDefinition id="jvmProcess">
    <jvmProcess class="org.objectweb.proactive.core.process.JVMNodeProcess"/>
   </processDefinition>
  </processes>
 </infrastructure>
</ProActiveDescriptor>
```

Fig. 1. Descriptor Example

3 ProActive File Transfer Model

We believe it is important to support File Transfer at different stages of Grid usage. For this, we have identified that File Transfer may occur:

- (I) Before application deployment;
- (II) After application deployment but before the user application is launched;
- (III) During user application;
- (IV) After the user application has finished.

We have defined *File Transfer Deploy* as the file transfer that can take place at (I) or (II)[1], *File Transfer Retrieve* at (IV), and *User Application File Transfer* at (III).

Since each stage has its own functional and environmental requirements, we have segmented our approach to tackle these stages as follows: Cases (I) and (IV) are handled through the *File Transfer Definitions* in the *ProActive Deployment Descriptor* (described in the first part of this section: 3.1). Cases (II) and (III) are handled using the *ProActive File Transfer API Tools* (described in 3.2).

3.1 File Transfer Definitions

A File Transfer definition must contain at least the following information: *what* to transfer; *when* to transfer; *where* from/to; and *how*. To promote reusability, and properly integrate these definitions into ProActive, we have grouped this information into two parts. The first is the *Abstract Definition* which mainly focuses on answering the *what* question:

Abstract Definitions

- Must have a unique identifier.
- May contain several *Files* or *Directories* specified as a source name, and optionally a destination name.
- Can be referenced from the *VirtualNode Level* and/or directly from the *Process Level*.

The second group is the *Concrete Definition* which focuses on answering the *when*, *where* and *how* questions:

Concrete Definitions

- Must have an identifier as to when the File Transfer will take place (*Deploy, Retrieve*).
- May have a sequence of copy protocols that will be used to copy the files, until one succeeds.
- Can contain a reference to one or several *Abstract Definition's* unique identifier.
- Can contain source and destination specific information like: *prefix*, *hostname*, *username, filepath separator, etc.*
- Can be referenced only from the *Process Level*.

To properly integrate these definitions into the *ProActive Deployment Descriptor Model* they must be referenced at the corresponding abstraction level, as shown in Figure 2. Both definitions will then be processed in the *File Transfer Workshop* module, to produce the *CopyProtocol* instances. The File Transfer

[1] Note that the File Transfer takes place after or before, but never in parallel with the job submission.

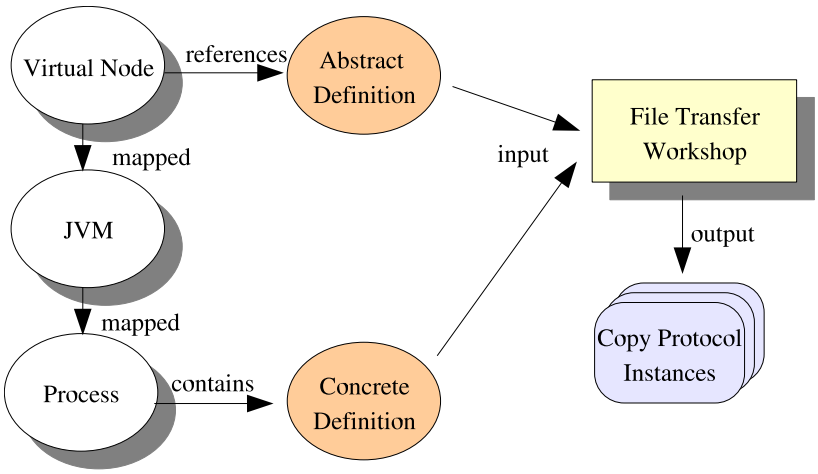


Fig. 2. ProActive File Transfer Model

will then take place by executing the *CopyProtocol* instances until one of them succeeds.

Automatically triggering *File Transfer Retrieve* after the user application has finished requires that we identify this particular application execution state. Currently ProActive does not yet provide the automatic identification of this state, and therefore, it is not currently feasible to automatically trigger the *File Transfer Retrieve* after the user application is finished. To solve this issue, we will provide the user with a specific API method (see section 3.2) for initiating the retrieval process.

3.2 ProActive File Transfer API Tools

For handling: *File Transfer Retrieve*, *User Application File Transfer*, or when no suitable CopyProtocol can be used at *File Transfer Deploy*, we believe it is useful to provide a *ProActive File Transfer API*:

ProActiveDescriptor.FileTransferRetrieve() Triggers the gathering of remote files. The necessary information for accomplishing this is assumed to be already specified in the XML Deployment Descriptor.

boolean ProActive.FileTransfer.push(String src, Node B, [String dst]) If node A invokes a *push* method on node B, then a file from A will be transfered to B.

File ProActive.FileTransfer.pull(String src, Node B, [String dst]) If node A invokes a *pull* method from node B, then a file from B will be transfered to A.

Since transferring a file can take a long time, we would like to continue with the asynchronism philosophy, systematically used in ProActive. In this philosophy, method calls invoked on active objects are asynchronous with transparent *future objects* and the synchronization is handled on a *wait-by-necessity* basis[3]. In

the same fashion, we can envision having *future files* where the user code may continue executing while the file transfer takes place, or until it is absolutely necessary to wait (*wait-by-necessity*).

4 File Transfer in ProActive Descriptor Deployment

4.1 File Transfer Related XML Tags

The *ProActive Deployment Descriptor Model* is specified with an XML schema. To add the File Transfer support we have included new XML tags to the schema, which correspond to the *Abstract* and *Concrete Definitions* discussed earlier (see section 3.1).

Figure 3 shows the XML tags which correspond to the File Transfer *Abstract Definitions*. The source and destination names are defined inside a *FileTransfer Definition* tag. Inspired by the UNIX *cp* command, if no destination name is specified, the source name is maintained.

```
....
</deployment>
<FileTransferDefinitions>
  <FileTransfer id="123">
    <file src="input.dat" dest="input.dat" />
    <file src="code.jar" />
    <dir src="exampledir" dest="otherdir"/>
  </FileTransfer>
  <FileTransfer id="456">
    <file src="output.dat"/>
  </FileTransfer>
</FileTransferDefinitions>
<infrastructure>
....
```

Fig. 3. File Transfer Definition

Figure 4 shows the File Transfer related attributes which can be specified at the VirtualNode abstraction level. These attributes are references to the *File Transfer Definitions* specified in Figure 3.

```
<VirualNode name="example" FileTransferDeploy="123"/>
```

Fig. 4. VirtualNode Level

Figure 5 shows the XML tags which correspond to the File Transfer *Concrete Definitions* (section 3.1). Inside the process, the FileTransfer tag becomes an el-

ement instead of an attribute. This happens because FileTransfer information is deployment *process* specific. In the example, an `implicit` value for the `refid` attribute in the `FileTransferDeploy` tag means that the *File Transfer Definition* reference is inherited from the VirtualNode definition (Figure 4), while the `FileTransferRetrieve` tag references directly a *File Transfer Definition* (Figure 3).

The `copyProtocol` sequence describes which protocols, and in what order, should be tried to transfer the files. Note the keyword `processDefault`, which corresponds to a default CopyProtocol related with the process.

```
<processDefinition id="xyz">
  <sshProcess>...
    <FileTransferDeploy refid="implicit">
      <copyProtocol>processDefault, scp, rcp</copyProtocol>
      <sourceInfo prefix="/home/user" />
      <destinationInfo prefix="/tmp"/>
    </FileTransferDeploy>
    <FileTransferRetrieve refid="456">
      <copyProtocol>processDefault</copyProtocol>
      <sourceInfo prefix="/tmp" hostname="foo.bar" username="smith"/>
      <destinationInfo prefix="/home/user"/>
    </FileTransferRetrieve>
  </sshProcess>
</processDefinition>
```

Fig. 5. Process Level

The flexibility of the proposed approach requires that the File Transfer is defined at two levels, but the user can use three. In the example, the *FileTransfer Deploy* is defined using three levels, while the *File Transfer Retrieve* uses only two. The main advantage of using three levels is the reusability of the process section. Several VirtualNodes can be mapped on to the same process, while transferring different files, by changing the *File Transfer Definition* reference at the VirtualNode level.

4.2 File Transfer CopyProtocols

As discussed in section 3, CopyProtocols are generated as the result of merging the abstract and concrete representations of File Transfer. In our current implementation, the user may specify an ordered list of CopyProtocols. Each CopyProtocol will be tried until one succeeds in transferring the files or all fail.

To clearly identify which CopyProtocols can be used with a certain deployment process, we have identified that these CopyProtocols can be of two different types:

Internal CopyProtocol Process dependant. The file transfer takes place at the same time as the job submission (process deployment). For example: Unicore or Globus file transfer.

External CopyProtocol Process independent. The file transfer takes place before the job submission. For example: *scp, rcp.*

Therefore, *Internal CopyProtocols* can only be used if deploying with the corresponding process. That is to say, Unicore file transfer can only be used when submitting a job to a Unicore site, while *External CopyProtocols* can be used independently of the type of job submission used. For example, with our current implementation, it is possible to transfer files with *scp (*or any other *External CopyProtocol)* to a Unicore site. This provides a flexible and powerful way of combining File Transfer and deployment tools.

5 Results

In this section we provide benchmarks on the current File Transfer implementation. The configuration was heterogeneous, mainly using a different site, located on a different network for each protocol. For the client machine we used a Xeon 2.0GHz with 1GB RAM. For the site machines we used a: SSH 3.0GHz/2GB RAM in the INRIA network (100Mbit/sec), Unicore 3.0GHz/1GB RAM on the Internet (18 hops, 51ms ping), and a Nordugrid site on the Internet (17 hops, 66ms ping).

5.1 On-the-fly Deployment

Usually, before a middleware deployment can take place, a preliminary configuration or installation is required. This preliminary stage can be something such as: environment configuration, remote middleware installation, libraries installation, or language interpreter installation. On-the-fly deployment corresponds to the capability of deploying without requiring this previous configuration or installation stage. In the case of ProActive, two main requirements must be fulfilled: library installation (client with its dependencies) and the Java Runtime Environment (JRE)[6].

For this benchmark we tested on-the-fly deployment with Unicore[4] and Nordugrid[7]. Both of them use the concept of *jobspace*: a specific file space created at job submission time, and destroyed when the job is finished. Therefore, no preliminary stage is possible. The File Transfer information was specified using the XML Deployment Descriptor, as specified in section 4.

Table 1 shows the required time (average of three) to deploy on these sites. For the Unicore site, we first deployed using an installed JVM transferring the ProActive libraries on-the-fly. In a second experience with Unicore, we transfered the ProActive libraries and a JRE, which were used to execute test application. For the Nordugrid site, we deployed transferring the ProActive libraries and a JRE at the same time. Note that the specified time for Nordugrid only considers the transferring of the ProActive libraries, since the client does not provide this information for the JRE.

Deployment Protocol	CopyProtocol	Requirements	Time [s]
Unicore	unicore	ProActive (3 MB)	7.013
Unicore	unicore	ProActive (3 MB) + JRE (16 MB)	138.234
Nordugrid	nordugrid	ProActive (3 MB) + JRE (16 MB)	14.012

Table 1. On-the-fly Deployment Benchmarks

5.2 FileTransfer Deploy

Since each protocol was benchmarked using a different server located in a different network in the Internet, it was not our objective to compare different deployment protocols. On the contrary, our objective was to measure the impact of different CopyProtocols configurations for the same deployment protocol.

Deployment Protocol	CopyProtocol Sequence	Time [s]
SSH -> LSF	`rcp`	0.566
SSH -> LSF	`scp`	0.659
SSH -> LSF	`rcp, scp`	0.911
Unicore	`scp`	2.449
Unicore	`processDefault`	8.018
Unicore	`scp, processDefault`	9.019
Unicore	`scp, rcp, processDefault`	11.022

Table 2. File Transfer Deploy Benchmarks

Table 2 shows the time required (average of three) to transfer a 1MB file at deployment time, using the protocols specified in the *CopyProtocol Sequence* column. Only the last protocol is configured to succeed but all of them are tried sequentially in the specified order. The *Deployment Protocol* column contains the deployment protocols used, and the "->" symbol represents a deployment using a chain of protocols. For example, *SSH->LSF* means that the Secure Shell protocol was used in combination with LSF to submit the job. Note that `processDefault` is a keyword for the default process CopyProtocol (see section 4.2).

6 Conclusions and Future Work

The results show that the proposed *File Transfer Model* has added important features to ProActive, namely: on-the-fly deployment, integration of third party File Transfer tools, and high user configurability of the File Transfer mechanism. Moreover the benchmarks show these features are useful and effectively working.

We believe the proposed *File Transfer Model* provides a flexible and useful Grid tool. Firstly, by clearly identifying the abstract and concrete aspects of File

Transfer, thus providing separation of concerns and reusability (3.1). Secondly, by addressing File Transfer at two key moments: *deployment* and *retrieval* time (4.1). Thirdly, by combining in a flexible way: job deployment protocols and File Transfer protocols (4.2). Finally, by providing asynchronous File Transfer tools during user application (3.2).

As future work we would like to:

- Implementation of the ProActive FileTransfer API.
- Continue the integration of third party File Transfer protocols into the *ProActive File Transfer Model.*

References

1. K. Davis, T. Goodale, A. Merzky. *Gat API Specification: Object Based.* `http://www.gridlab.org/WorkPackages/wp-1/documentation.html`
2. F. Baude, D. Caromel, F. Huet, L. Mestre and J. Vayssiere. *Interactive and Descriptor-based Deployment of Object-Oriented Grid Applications. pp. 93-102, in HPDC-11*, Edinburgh, Scotland, July 2002.
3. D. Caromel. *Towards a Method of Object-Oriented Concurrent Programming.* Communications of the ACM, 36(9):90-102, September 1993.
4. D. Erwin, editor. *UNICORE plus final report – uniform interface to computing resources.* Forschungszentrum Julich 2003, ISBN 3-00-011592-7.
5. T. Kielmann, A. Mersky, H. Bal, F. Baude, D. Caromel, F. Huet. *Grid Application Programming Environments.* CoreGRID Technical Report Number TR-0003. June 21, 2005.
6. Java Runtime Evirorment `http://www.java.com`
7. P. Eerola, T. Ekelof, M. Ellert, J. R. Hansen, A. Konstantinov, B. Konya, J. L. Nielsen, F. Ould-Saada, O. Smirnova, A. Waananen. *The NorduGrid architecture and tools.* Proceedings of CHEP 2003, eConf C0303241:MOAT003,2003.
8. Load Sharing Facility `http://www.platform.com/`
9. Portable Batch System `http://www.openpbs.org/`
10. Sun Grid Engine `http://gridengine.sunsource.net/`
11. ProActive, `http://www-sop.inria.fr/oasis/ProActive/`
12. von Laszewski, G., Gawor, J., Plaszczak, P., Hategan, M., Amin, K., Madduri, R., and Gose, S. 2004. *An overview of grid file transfer patterns and their implementation in the Java CoG kit.* Neural, Parallel Sci. Comput. 12, 3 (Sep. 2004), 329-352.
13. von Laszewski, G., Alunkal, B., Gawor, J., Madhuri, R., Plaszczak, P. & Sun, X.-H. (2003). A File Transfer Component for Grids, in H. Arabnia & Y. Mun (eds), Proceedings of the International Conferenece on Parallel and Distributed Processing Techniques and Applications, Vol. 1, CSREA Press, Las Vegas, pp. 24-30.

GRID superscalar enabled P-GRADE portal

Róbert Lovas[1], Raül Sirvent[2], Gergely Sipos[1],
Josep M. Pérez[2], Rosa M. Badia[2], Péter Kacsuk[1]

[1]Computer and Automation Research Institute, Hungarian Academy of Sciences (MTA SZTAKI)
{rlovas, sipos, kacsuk}@sztaki.hu

[2]Barcelona Supercomputing Center and UPC, SPAIN
{rsirvent, rosab, perez}@ac.upc.edu

Abstract. One of the current challenges of the Grid scientific community is to provide efficient and user-friendly programming tools. GRID superscalar allows programmers to write their Grid applications as sequential programs. However, on execution, a task-dependence graph is built and the inherent concurrency of the task is exploited and executed in a Grid. P-GRADE Portal is a workflow-oriented grid portal with the main goal to cover the whole lifecycle of workflow-oriented computational grid applications. In this paper the authors discuss the different options taken into account to integrate these two frameworks.

1 Introduction

One of the issues that raises current interest in the Grid community and in the scientific community in general is the application programming in Grids. While more and more scientific groups aims to use the power of the Grids, the difficulty of porting applications to the Grid (what sometimes is called application "gridification") may be an obstacle to the adaptation of this technology.

Examples of efforts for provide Grid programming models are ProActive, Ibis, or ICENI. ProActive [15] is a Java library for parallel, distributed and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API masking the specific underlying tools and protocols used, and allowing to simplify the programming of applications that are distributed on a LAN, on a cluster of PCs, or on Internet Grids. The library is based on an active object pattern, on top of which a component-oriented view is provided.

The Ibis Grid programming environment [16] has been developed to provide parallel applications with highly efficient communication API's. Ibis is based on the Java programming language and environment, using the "write once, run anywhere" property of Java to achieve portability across a wide range of Grid platforms. Ibis aims at Grid-unaware applications. As such, it provides rather high-level communication API's that hide Grid properties and fit into Java's object model.

ICENI [17] is a grid middleware framework with an added value to the lower-level

grid services. It is a system of structured information that allows to match applications with heterogenous resources and services, in order to maximise utilisation of the grid fabric. Applications are encapsulated in a component-based manner, which clearly separates the provided abstraction and its possibly multiple implementations. Implementations are selected at runtime, so as to take advantage of dynamic information, and are selected in the context of the application, rather than a single component. This yields to an execution plan specifying the implementation selection and the resources upon which they are to be deployed. Overall, the burden of code modification for specific grid services is shifted from the application designer to the middleware itself.

Tools as the P-GRADE Portal or GRID superscalar aims to ease the utilization of the Grid, but cover different areas from an end-user's point of view. While P-GRADE Portal is a graphical-based tool, GRID superscalar is based on imperative language programs. Although there is some overlap in functionality, both tools show a lot of complementarity and it is very challenging to make them inter-operable. The integration of these tools may be a step towards achieving the idea of the "invisible" Grid for the end-user.

This work has been developed in the context of the NoE CoreGRID. More specifically, in the virtual institute "Systems, Tools and Environments" (WP7) and aims to contribute to the task 7.3 "Integrated Toolkit". The "Integrated Toolkit" will provide means to develop Grid-unaware applications, for execution in the Grid in a way transparent to the user and increasing the performance of the application.

In this paper the integration of the P-GRADE Portal and the GRID superscalar is discussed. In Section 2 the P-GRADE Portal is presented and Section 3 covers the description of the GRID superscalar framework. Then in Section 4 a comparison between both tools is given. Following that, Section 5 discusses an integration solution, and at the end of this paper Section 6 presents some conclusions, related work and future work.

2 P-GRADE Portal

The P-GRADE Portal [1] is a workflow-oriented grid portal with the main goal to cover the whole lifecycle of workflow-oriented computational grid applications. It enables the graphical development of workflows consisting of various types of executable components (sequential, MPI or PVM programs), executing these workflows in Globus-based grids relying on user credentials, and finally analyzing the correctness and performance of applications by the built-in visualization facilities.

A P-GRADE Portal workflow is an acyclic dependency graph that connects sequential and parallel programs into an interoperating set of jobs. The nodes of such a graph are jobs, while the arc connections define the execution order of the jobs and the data dependencies between them that must be resolved by the workflow manager during the execution. An ultra-short range weather forecast (nowcast) grid application [2] is shown in Fig 1 as an example for a P-GRADE Portal workflow.

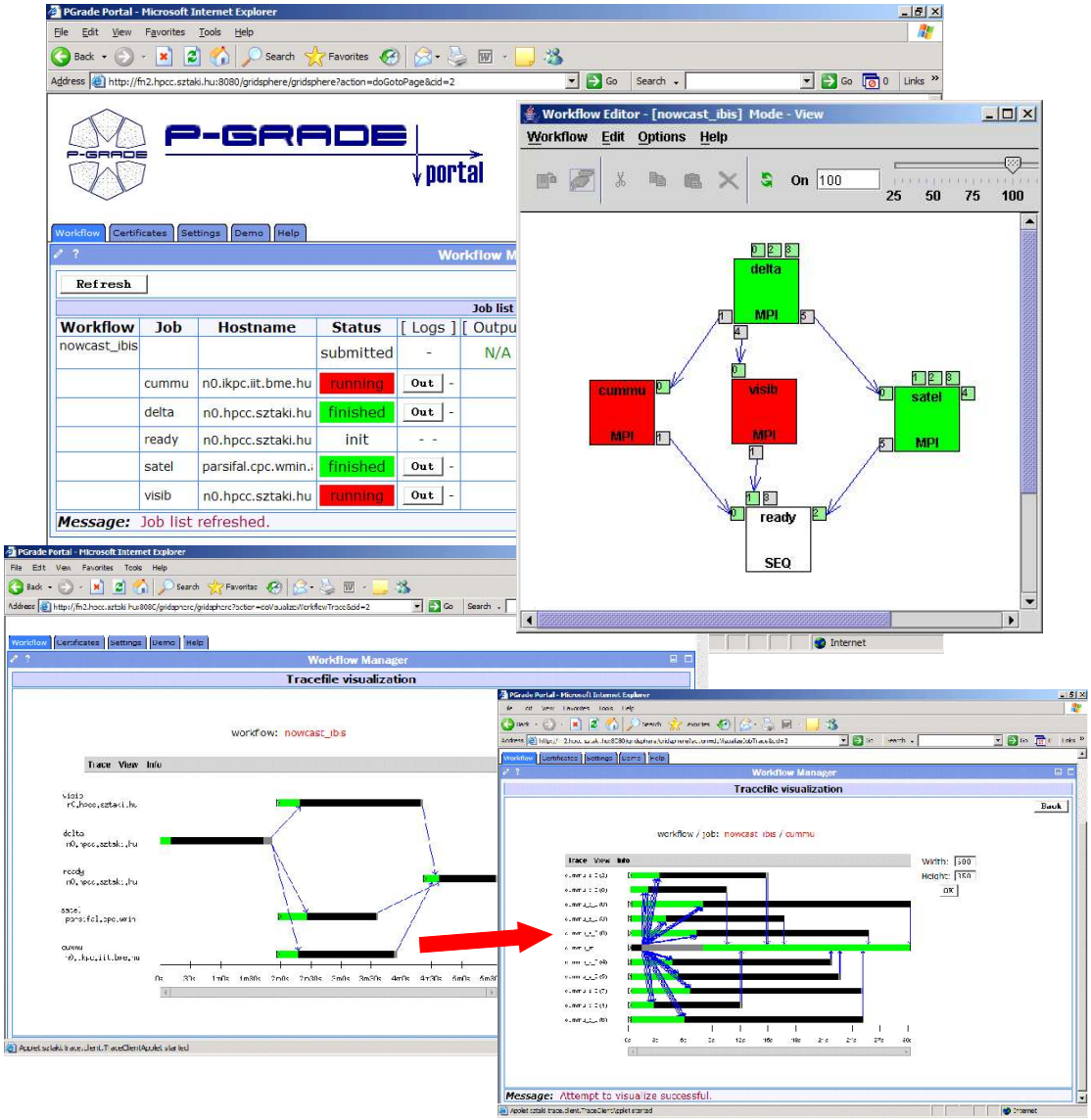


Fig. 1: Meteorological application in P-GRADE Portal; workflow manager and workflow description with status information, multi-level visualization of a successful execution

Nodes (labelled as *delta*, *cummu*, *visib*, *satel* and *ready* in) represent jobs while rectangles (labelled by numbers) around the nodes are called ports and represent data files that the corresponding jobs expect or produce. Directed arcs interconnect pairs of input and output files if an output file of a job serves as an input file for another job. The semantics of the workflow execution means that a job (a node 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 the job have successfully terminated, and all the user-defined input files are available either on the portal server and at the pre-defined grid storage providers. 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 DAGMan [3], the workflow manager used in the Portal transfers these files – together with the binary executable – to the site where the job has been allocated by the developer for execution. Managing the transfer of files and recognition of the availability of the necessary files is the task of the workflow manager subsystem.

To achieve high portability among the different grids, the P-GRADE Portal has been built onto the GridSphere portal framework [14], and the Globus middleware,

and particularly those tools of the Globus Toolkit that are generally accepted and widely used in production grids today. GridFTP, GRAM, MDS and GSI [4] have been chosen as the basic underlying toolset for the Portal.

GridFTP services are used by the workflow manager subsystem to transfer input, output and executable files among computational resources, among computational and storage resources and between the portal server and the different grid sites. GRAM is applied by the workflow manager to start up jobs on computational resources. An optional element of the Portal, the information system portlet, queries MDS servers to help developers map workflow components (jobs) onto computational resources. GSI is the security architecture that guarantees authentication, authorization and message-level encryption facilities for GridFTP, GRAM and MDS sites.

The choice of this infrastructure has been justified by connecting the P-GRADE Portal to several grid systems like the GridLab test-bed, the UK National Grid Service, and two VOs of the LCG-2 Grid (See-Grid and HunGrid VOs). Notice, that most of these grid systems use some extended versions of the GT-2 middleware. The point is that if the compulsory GRAM, GridFTP and GSI middleware set is available in a VO, then the P-GRADE Portal can be immediately connected to that particular system.

Currently, the main drawback of P-GRADE portal is the usage of Condor DAGMAN as the core of workflow manager, which cannot allow the user to create cyclic graphs.

3 GRID superscalar

The aim of GRID superscalar [5] is to reduce the development complexity of Grid applications to the minimum, in such a way that writing an application for a computational Grid may be as easy as writing a sequential application [6]. It is a new programming paradigm for Grid-enabling applications, composed of an interface, a run-time and a deployment center. With GRID superscalar a sequential application composed of tasks of a certain granularity is automatically converted into a parallel application where the tasks are executed in different servers of a computational Grid.

Fig. 2 outlines GRID superscalar behaviour: from a sequential application code, a task dependence graph is automatically generated, and from this graph the runtime is able to detect the inherent parallelism and submit the tasks for execution to resources in a grid.

The interface is composed by calls offered by the run-time itself and by calls defined by the user. The main program that the user writes for a GRID superscalar application is basically identical to the one that would be written for a sequential version of the application. The differences would be that at some points of the code, some primitives of the GRID superscalar API are called. For instance, GS_On and GS_Off are respectively called at the beginning and at the end of the application. Other changes would be necessary for those parts of the program where files are read or written. Since the files are the objects that define the data dependences, the run-time needs to be aware of any operation performed on them. The current version offers four primitives for handling files: GS_Open, GS_Close, GS_FOpen and GS_FClose. Those primitives implement the same behavior as the standard open,

close, fopen and fclose functions. In addition, the GS_Barrier function has been defined to allow the programmers to explicitly control the tasks' flow. This function waits until all Grid tasks have finished. Also the GS_Speculative_End function allows an easy way to implement parameter studies by dealing with notifications from the workers in order to stop the computation when an objective has been reached. It is important to point that several languages can be used when programming with GRID superscalar (currently C/C++, Perl, Java and Shell script are supported).

Besides these changes in the main program, the rest of the code (including the user functions) do not require any further modification.

The interface defined by the user is described with an IDL file where the functions that should be executed in the Grid are included. For each of these functions, the type and direction of the parameters must be specified (where direction means if it is an input, output or input/output parameter). Parameters can be files or scalars, but in the current version data dependencies will only be considered in the case of files.

The basic set of files that a programmer provides for a GRID superscalar application are a file with the main program, a file with the user functions code and the IDL file. From the IDL file another set of files are automatically generated by the code generation tool *gsstubgen*. This second set of files are stubs and skeletons that converts the original application into a grid application that calls the run-time instead of calling the original functions. Finally, binaries for the master and workers are generated and the best way to do this it by using the GS *deployment center*.

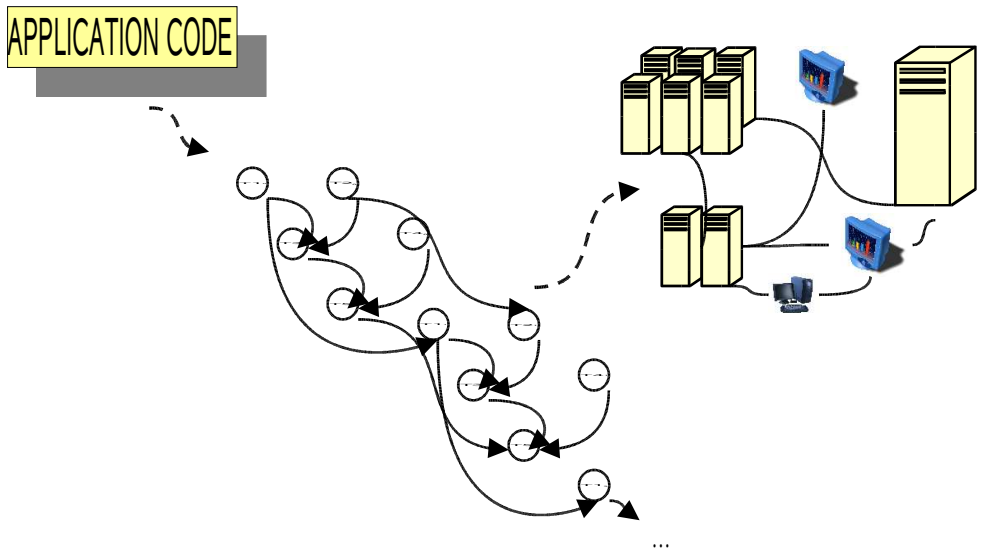


Fig. 2: GRID superscalar behavior

The GS deployment center is a Java based Graphical User Interface. Is able to check the grid configuration and also performs an automatic compilation of the main program in the localhost and worker programs in the server hosts.

GRID superscalar provides an underlying run-time that is able to detect the inherent parallelism of the sequential application and performs concurrent task submission. The components of the application that are objective of this concurrency exploitation are the functions listed in the IDL file. Each time one of these functions is called, the runtime system is called instead of the original function. A node in a data-dependence graph is added, and file dependencies between this function and functions called previously are detected. From this data-dependence graph, the

runtime can submit for concurrent execution those functions that do not have any dependence between them. In addition to a data-dependence analysis based on those input/output task parameters which are files, techniques such as file renaming, file locality, disk sharing, checkpointing or constraints specification with ClassAds [7] are applied to increase the application performance, save computation time or select resources in the Grid. The run-time has been ported to different grid middlewares and the versions currently offered are: Globus 2.4 [8], Globus 4 [8], ssh/scp and Ninf-G2 [9].

Some possible limitations in current version of GRID superscalar are that only give support to a single certificate per user at execution, and also that monitoring is only provided by log messages (text format), although a graphical monitoring interface is being developed using uDrawGraph [10]. Regarding resource brokering, the selection is performed inside the run-time, but resource discovery is not supported, and machines are specified statically by the user using the GS *deployment center*. During the execution of the application the user can change the machine's information (add, remove or modify hosts parameters). Performance analysis of the application and the run-time has been done using Paraver [11], but is not currently integrated in the runtime in such a way that end-users can take benefit from it.

4 Comparison of P-GRADE Portal and GRID superscalar

The aim of both the P-GRADE Portal and the GRID superscalar systems is to ease the programming of grid systems, by providing high-level environments on top of the Globus middleware. While the P-GRADE Portal is a graphical interface that integrates a workflow developer tool with the DAGMan workflow manager systems, the GRID superscalar is a programming API and a toolset that provide automatic code generation, as well as configuration and deployment facilities. The following table outlines the differences between both systems:

Products/ Functionalities	GRID superscalar	P-GRADE portal
Support for data parallelism (graph generation)	**Advanced** automatic detection of data parallelism	**Manual** user has to express explicitly
Support for acyclic/conditional dataflows	**YES** using C or PERL	**NO** based DAGMAN/Condor
Compilation & staging of executables	**YES** Deployment Center	**Limited** **only run-time staging is supported**
Thin client concept	**NO** Globus client and full GS installation are needed	**YES** only a Java-enabled browser required
Monitoring & performance visualization	**NO** Debug/log messages are available	**YES** multi-level visualization: workflow/job/processes

Multi-Grid support	**NO** only one certificate	**YES** several certificates are handled at the same time using myproxy server
Support for existing MPI/PVM applications	**Limited** by using "wrapper" technology	**YES** MPI/PVM jobs or GEMCLA services

5 Overview of the solution

The main purpose of the integration of the GRADE Portal – GRID superscalar system is to create a high level, graphical grid programming, deployment and execution environment that combines the workflow-oriented thin client concept of the P-GRADE Portal with the automatic deployment and application parallelisation capabilities of GRID superscalar. This integration work can be realised in three different ways:

- *Scenario 1*: A new job type can be introduced in P-GRADE workflow for a complete GRID superscalar application.
- *Scenario 2*: A sub-graph of P-GRADE workflow can be interpreted as a GRID superscalar application.
- *Scenario 3*: A GRID superscalar application can be generated based on the entire P-GRADE workflow description.

In case of the first two scenarios, the interoperability between the existing P-GRADE workflow applications and GRID superscalar applications would be provided by the system. On the other hand, Scenario 2 and 3 would enable the introduction of new language elements into P-GRADE workflow description for steering the data/control flow in a more sophisticated way; e.g. using conditional or loop constructs similarly to UNICORE [13]. Scenario 3 was selected as the most promising one and in this paper is discussed in detail.

Before the design and implementation issues, it is important to distinguish the main roles of the site administrators, developers, and end-users which are often mixed and misunderstood in academic grid solutions. The new integrated system will support the following actors (see Fig. 3);

1. The *site administrator*, who is responsible for the installation and configuration of the system components such as P-GRADE portal, GRID superscalar, and the other required grid-related software packages.
2. The Grid *application developer and deployer*, who develops the workflow application with the editor of P-GRADE portal, configures the access to the Grid resources, and deploys the jobs with GS deployment center, and finally optimizes the performance of the application using Mercury Grid monitor and the visualisation facilities of P-GRADE portal.
3. The *end-user*, who runs and interprets the results of the executions with P-GRADE portal and its application-specific portlets from any thin client machine.

Therefore, there are several benefits of the integrated solution from the end-users' points of view; they do not have to tackle the grid related issues.

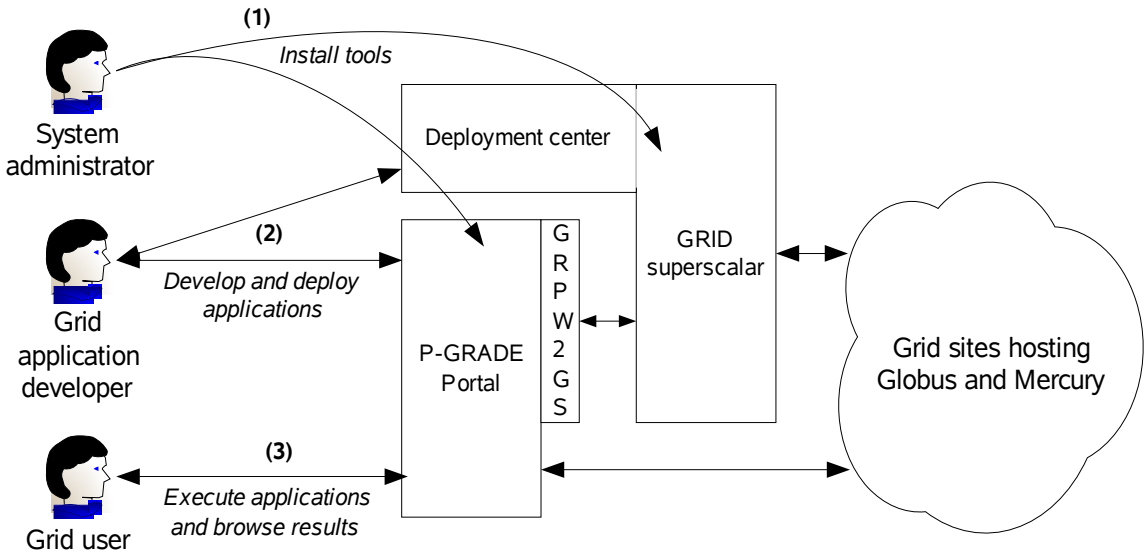


Fig. 3: The roles in the integrated P-GRADE Portal – GRID superscalar system

In order to achieve these goals a new code generator GRPW2GS is integrated in P-GRADE portal. It is responsible for the generation of a GRID superscalar-compliant application from a workflow description (GRPW): an IDL file, a main program file, and a functions file.

In the ***IDL file***, each job of the actual workflow is listed as a *function* declaration within the interface declaration. An example of generated GRID superscalar IDL file is shown in next lines:

```
interface workflowname {
        void jobname (dirtype File filename,  …);
        …
};
```

where *workflowname* and *jobname* are unique identifiers, and inherited from the workflow description. The *dirtype* can be **in** or **out** depending to the direction of the type of the actual file. The actual value of *filename* must depend on the dependencies of the file. If it is a file without dependencies (i.e. input or output of the entire workflow application), the *filename* can be the original name. On the other hand, if the file is an input of another job, a unique file identifier is generated since in P-GRADE descriptions the filenames are not unique at workflow level.

The following lines shows the structure of a main program file generated based from a workflow.

```
#include "GS_master.h"
    void main(int argc, char **argv) {
            GS_On();
            jobname1("filename1", …);
            jobname2("filename2", …);
            …
            GS_Off(0);
    }
```

For the generation of the functions file, two options have been taken into

consideration; (1) using a simple wrapper technique for legacy code, or (2) generating the entire application from source.

In the first case, the executable must be provided and up-loaded to the portal server by the developer similarly to the existing P-GRADE portal solution. The definitions of function calls (corresponding to the jobs) in the functions file contain only system calls to invoke these executables, which are staged by the P-GRADE portal to the appropriate site (selected by the resource broker).

In the second case, the application developer uploads the corresponding C code as the 'body' of the function using the job properties dialogue window of P-GRADE portal. In this case, the developer gets a more flexible and architecture-independent solution, since the GS deployment center can assist to create the appropriate executables on Globus sites with various architectures.

After the automatic generation of code, the application developer can deploy the code by GS deployment center, and the performance analysis phase can be started. For this purpose, the execution manager of GRID superscalar has to generate a Prove-compliant trace file to visualise the workflow-level execution. It means the instrumentation of its code fragments by GRM, which are dealing with the resource selection, job submission and file transfers. In order to get a more detailed view, the parallel MPI code can be also instrumented by a PROVE-compliant MPICH instrumentation library developed by SZTAKI.

Concerning the resource broker; the job requirement (defined in the job attributes dialog window for each jobs) can be also passed to the GS broker from the workflow editor in case of GT-2 grids, or the LCG-2 based resource broker can be also used in P-GRADE portal.

6 Conclusions, related and future work

The paper presented an initial solution for the integration of P-GRADE portal and GRID superscalar. The solution is based on the generation of a GRID superscalar application from a P-GRADE workflow. The GS deployment center is also used to automatically deploy the application in the local and server hosts.

Concerning the future work, the prototype must be finalized, and then the addition of conditional and loop constructs, and support for parameter study applications at workflow level can be started in order to get high-level control mechanisms, similar to UNICORE [13].

Therefore, we will get closer a new toolset that can assist to system administrators, programmers, and end-users at each stage of software development, deployment and usage of complex workflow based applications on the Grid.

The integrated GRID superscalar – P-GRADE Portal system shows many similarities with the GEMLCA [12] architecture. The aim of GEMLCA is to make pre-deployed, legacy applications available as unified Grid services. Using the GS deployment center, components of P-GRADE Portal workflows can be published in the Grid for execution as well. However, while GEMLCA expects compiled and already tested executables, GRID superscalar is capable to publish components from source code.

Acknowledgments

This word has been partially supported by NoE CoreGRID (FP6-004265) and by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01.

References

1. G. Sipos and P. Kacsuk: Classification and Implementations of Workflow-Oriented Grid Portals, To appear in the Proc. Of HPCC-2005 Conference
2. R. Lovas, et al.: Application of P-GRADE Development Environment in Meteorology. Proc. of DAPSYS'2002, Linz,, pp. 30-37, 2002.
3. T. Tannenbaum, D. Wright, K. Miller, and M. Livny: Condor - A Distributed Job Scheduler. Beowulf Cluster Computing with Linux, The MIT Press, MA, USA, 2002.
4. I. Foster, C. Kesselman: Globus: A Toolkit-Based Grid Architecture, In I. Foster, C. Kesselmann (eds.) „The Grid: Blueprint for a New Computing Infrastructure", Morgan Kaufmann, 1999, pp. 259-278.
5. GRID superscalar Home Page, http://www.cepba.upc.edu/grid/
6. Rosa M. Badia, Jesús Labarta, Raül Sirvent, Josep M. Pérez, José M. Cela and Rogeli Grima, "Programming Grid Applications with GRID superscalar", Journal of Grid Computing, Volume 1, Issue 2, 2003.
7. M. Solomon: The ClassAd Language Reference Manual, http://www.cs.wisc.edu/condor/classad/
8. The Globus project, http://www.globus.org/
9. Ninf Project Home Page, http://ninf.apgrid.org/
10. uDraw(Graph), http://www.informatik.uni-bremen.de/~davinci/
11. PARAVER, http://www.cepba.upc.edu/paraver/
12. T. Delaittre, T. Kiss, A. Goyeneche, G. Terstyanszky, S.Winter, P. Kacsuk: GEMLCA: "Running Legacy Code Applications as Grid Services" To appear in Journal of Grid Computing, Vol. 3., No. 1, 2005.
13. Dietmar W. Erwin: "UNICORE - A Grid Computing Environment", Concurrency and Computation: Practice and Experience Vol. 14, Grid Computing environments Special Issue 13-14, 2002.
14. Jason Novotny, Michael Russell, Oliver Wehrens: GridSphere: a portal framework for building collaborations, Concurrency and Computation: Practice and Experience, Volume 16, Issue 5 , Pages 503–513, 2004
15. ProActive, see http://www-sop.inria.fr/oasis/ProActive
16. Rob V. van Nieuwpoort, Jason Maassen, Gosia Wrzesinska, Rutger Hofman, Ceriel Jacobs, Thilo Kielmann, Henri E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. Concurrency & Computation: Practice & Experience, Vol. 17, No. 7-8, pp. 1079-1107, 2005.
17. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T . Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. Parallel Computing, 28(12), 2002.

Towards Goal-Oriented Refinement of Grid Trust and Security Policies for Virtual Organizations

Philippe Massonet[1] and Alvaro Arenas[2]

[1] CETIC, rue Clément Ader, 8,
B-6041 Gosselies, Belgium
phm@cetic.be
http://www.cetic.be
[2] CCLRC Rutherford Appleton Laboratory, Chilton, Didcot
Oxon, UK, OX11 0QX
A.E.Arenas@rl.ac.uk
http://www.cclrc.ac.uk/

Abstract. This position paper reviews security and trust issues in the Grid and identifies the need for policy-based management of trust and security for virtual organisation management. It then identifies the need for designing such policies, and suggests a method for deriving Grid trust, security and privacy policies for virtual organisations from high-level properties. The method should enable the definition and refinement of high-level trust, security and privacy properties into a coherent set of abstract policy rules. The refined abstract policy rules can then be translated into concrete Grid trust, security and privacy policies for virtual organization (VO) management. The policies can be used during the different phases of VO management lifecycle.

1 Introduction

In the Internet world, trust has been recognised as an important aspect of decision making for electronic commerce [1, 2]. Customers must trust that sellers will provide the services they advertise, and will not disclose private customer information. Trust in the supplier's competence and honesty will influence the customer's decision as to which supplier to use. Sellers must trust that the buyer is able to pay for goods or services, is authorised to make purchases on behalf of an organisation or is not underage for accessing service or purchasing certain goods.

How is the situation in the Grid? Fundamental to the Grid definition is the idea of resource sharing [3]. The Grid was initiated as a way of supporting scientific collaboration, where many of the participants knew each other. In this case, there is an implicit trust relation, all partners have a common objective –for instance to realise a scientific experiment- and it is assumed that resources would be provided and used within some defined and respected boundaries. However, when the Grid is intended to be used for business purposes, it is necessary to share resources with unknown parties. Such interactions may involve some degree of risk since the resource user

cannot distinguish between high and low quality resource providers on the Grid. The inefficiency resulting from this asymmetry of information can be mitigated through trust mechanisms.

This article suggests a policy-based approach for handling trust, security and privacy issues at the virtual organization level. A method that builds upon requirements engineering refinement techniques is suggested as future work. Section 2 reviews current security mechanisms used in the Grid, and relates them to trust. Section 3 briefly describes VOs and describes the policy-based approach to VO management. Section 4 describes how a methodology for deriving operational policy rules from more abstract trust, security and privacy (TSP) objectives could be defined using existing requirements engineering methodologies.

2 Trust and Security Issues in the Grid

2.1 Trust and Security in Grids

Trust mechanisms have become complementary to security mechanisms. Security mechanisms typically protect resources from malicious users by restricting access to only authorised users. However, in many situations within distributed applications one has to protect oneself from those who offer resources so that the problem is in fact reversed. For instance, a resource providing information can act deceitfully by providing false or misleading information, and traditional security mechanisms are unable to protect against this type of threat. As noted in [2], trust systems can provide protection against such threats. The difference between these two approaches to security was first described by Rasmusson and Janssen in [4] who used the term hard security for traditional mechanisms like authentication and access control, and soft security for what they called social control mechanisms, of which trust is an example. These results build up from previous surveys on trust and security for distributed systems and the Grid [5, 6, 1, 2, 7, 8].

2.2 Traditional Security Mechanisms

Traditional trust and security technologies that play an important role in the Grid include mechanisms such as authentication, authorisation and confidentiality. A survey on these technologies can be found in [8].

Authentication deals with verification of the identity of an entity within a network. An entity may be a user, a resource or a service provided as part of the Grid. One of the technologies playing a central role in authentication is Public Key Infrastructure (PKI), which defines message formats and protocols that allow entities to securely

communicate claims and statements. The most used assertions are those that bind identity and attributes statements to keys. The most popular PKI is defined by the IETF's PKIX working group, which defines a security system used for identifying entities (users and resources) through the use of X.509 identity certificates. In this PKI, highly trusted entities know as certificate authorities (CA) issue X.509 certificates where essentially a unique identity name and the public key of an entity are bound through the digital signature of that CA.

Authorisation deals with the verification of an action that an entity can perform after authentication was performed successfully. In a Grid, resource owners will require the ability to grant or deny access based on identity, membership of groups or VOs, and other dynamic considerations. Thus policies must be established that determine the capabilities of allowed actions. Authorisation is close related to access control trust. A good description of the current state of authorisation in Grid computing appears in [9].

There are several architectural proposals for handling authorisation in Grids. One of the earliest attempts at providing authorisation in VOs was in the form of the Globus Toolkit Gridmap files. This file simply holds a list of the authenticated distinguished names of the Grid users and the equivalent local user account names that they are to be mapped into. Access control to a resource is then left up to the local operating system and application access control mechanisms. As can be seen, this neither allows the local resource administrator to set a policy for who is allowed to do what, nor does it minimise his/her workload. The Community Authorisation Service (CAS) [10] was the next attempt by the Globus team to improve upon the manageability of user authorisation. CAS allows a resource owner to grant access to a portion of his/her resource to a VO (or community hence the name CAS), and then let the community determine who can use this allocation. The resource owner thus partially delegates the allocation of authorisation rights to the community.

However, some authors have identified limitations in CAS-based authorisations mechanisms and the need for a fine-grain authorization system [11], so that authorization services, such as Akenti [12] and Permis, have been integrated in the Globus Toolkit. One of the limitations is that Grid resource access is considered an atomic operation, and no further controls are executed after access to a resource is granted.

Concerning **confidentiality**, the data being processed in a Grid may be subject to considerable confidentiality constraints, either due to privacy concerns or issues of intellectual property. As mentioned in [6], confidentiality is usually associated with the encryption of data only, however there are other aspects to be considered for the case of Grids. The use of Grids implies that confidential data is stored in online accessible databases. Access to their interfaces must be carefully controlled, both to allow access only to appropriate users, and also to allow queries and simulations to run over these highly confidential data without that data being compromised or revealed. Confidentiality also extents to the privacy requirements of the actual users and resources.

3 Policy-Based Trust and Security Management of VOs

3.1 Trust Security and Privacy in Virtual Organisations

A virtual organization may be defined as a set of resources, users and rules governing the sharing of the resources. One of the main issues in VOs is how to allow members of a VO to access shared resources in an easy, transparent, coordinated, trusted and secure way.

These issues have to be dealt with during all phases of the VO lifecycle. In an initial phase the VO has to be created, and initial members must be identified. During VO operation members offer and use services within the VO. The VO may also evolve while it is operating, and the global rules governing it could change. When the purpose of the VO has been achieved, the VO can be dissolved.

A primary step in implementing an electronic system is identifying systems aspects such as security goals and risks. It is important to establish who the authorized users might be, how they will access the system and data, how unauthorised users will be denied access, and how data will be protected within the organizations as well as outside the organization.

A security policy must address organization’s specific risks. To understand risks, an appropriate player should perform a security audit that identifies vulnerabilities and rates both the severity of each threat and its likelihood of occurring. Virtual organizations offer several areas of risk due the involvement of various parties and the fact that VOs traverse multiple domains and hosting environments.

An important component in a VO is the VO Management subsystem, responsible for coordinating all functionalities of the VO. Most of the VO management systems are limited to membership management. However, this functionality is quite limited if one wants to exploit the Grid for business. VO management should be connected to (or include), among other things, facilities related to contract management, facilities such as the management of the collaboration agreement specifying the "rules of the games" under which all VO members agreed to participate in the VO. Such rules of the games correspond to policies that could be derived from the goals of the VO [13].

If TSP policies are to be enforced, the VO management must be able to reason about Trust and Security Properties in Dynamic Virtual Organisations. This covers reasoning about composition of Grid services taking into consideration trust and security properties as well as non-functional properties such as Quality of Service. It also requires techniques and tools to verify that all VO participants along the VO life cycle respect the general properties of a VO.

3.2 Trust, Security and Privacy Policy-based approach to VO Management

If policy-based approaches for VO management are to be adopted, it is important to develop methods for performing analysis and refinement of policy specifications. Policy refinement is the process of deriving operational policies from high-level goals, i.e. expressed in terms of actions on the underlying VO. These operations must be available on the underlying VO, and must be able to satisfy the goals [14].

Policy based approaches to VO management are of particular importance because they allow the separation of the rules that govern the behaviour of a VO from the functionality provided by the VO. This means that it is possible to adapt the behaviour of a VO without the need to recode functionality, and changes can be applied without stopping the VO.

Many issues related to policy-based VO management are still research questions and the work described here plans to tackle some of them. For instance, if a policy defined in a contract or SLA has been broken what action should the VO manager take? Adaptation policies - also sometimes referred to as Obligation policies are in essence event-condition-action rules that determine how components should adapt in response to events arising as part of a VO. The SLA or contract may state penalty clauses that apply under such circumstances, if so, should such clauses be automatically enacted – should VO members be fined, or removed from the VO automatically? Although it is technically feasible to do this, it is not believed that the market is yet ready for such dramatic automation. Rather it is assumed that the human who manages the VO should be notified and human action may be required. This is an assumption based on discussions with those involved in current VO, and other outsourcing relationships. As the technology matures and the market becomes more accustomed to it, the opportunity for increased automation in actions is expected to increase, but at present the more conservative option is judged to be appropriate for the market.

4 Towards a Refinement Method of Trust, Security and Privacy Properties into VO Policies

4.1 Deriving VO TSP Policies from TSP Objectives

Designing a coherent set of policy rules that meets all trust, security and privacy goals for VO management and VO through all phases of its lifecycle is a not a trivial task. Modifying the policy during VO evolution can in no way disrupt the ongoing operation of the VO. Changes to the policies must guarantee that there is no or minimal disruption of service.

Designing such policies and making them evolve cannot be done in a trial and error manner. A method is needed to help design such policies to meet predefined trust, security and privacy goals. This section briefly outlines how some results from goal-oriented requirements engineering [16] could be applied to design Grid trust, security and privacy policies for virtual organization (VO) management.

A policy-based virtual organisation management system should allow the description of high-level policies, enable their refinement into lower-level ones and map them to commands that ultimately configure the managed resources. The general "on-event and if condition then action" structure of policy rules makes it possible to consider policy-based systems as event/state-driven systems and use formal methods to analyze their behaviour [15]. At the Grid application level, the challenge is how to define business level TSP properties and guarantee that the TSP polices used during the VO lifecycle will always satisfy the desired TSP properties.

Such a methodology can build on existing requirements engineering methodologies that have been applied to security properties [16], and also to Grid applications [17]. This methodology can support the refinement of TSP properties into TSP policies [15]. This requires expressing and reasoning about TSP properties and VO topologies. The methodology should support the refinement of TSP properties into requirements for different classes of VO (topologies), and then into TSP policies. An abstract policy language could be used, so that it can be mapped to a concrete Grid policy language for VO management.

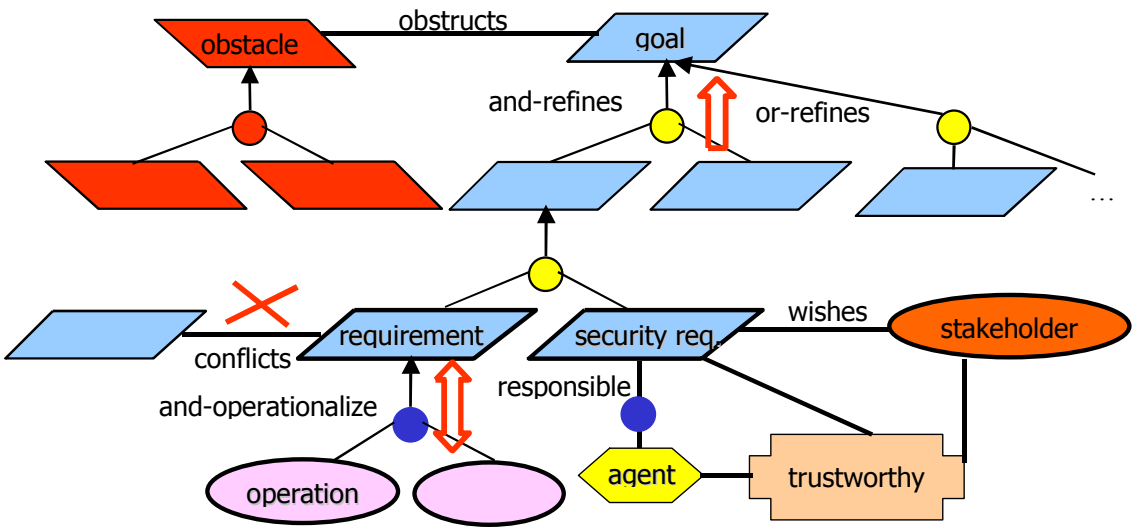


Figure 1 Goal modeling meta-concepts

Figure 1 shows the main goal-oriented requirement meta-concepts needed to model, trust, security and privacy goals. Functional and non-functional goals can be and-refined into sub-goals, and into requirements. And-refinement means that the conjunction of the sub-goals must imply the parent goal. Goals may also be or-refined allowing alternative ways of refining goals to be expressed. Obstacles to goals need to be identified and refined into sub-obstacles until they can be handled. Requirements are goals that can be assigned to the responsibility of a single agent. This agent needs

to be capable of performing operations so that the requirement is met. This is called operationalizing a requirement and means that the definition of the operations (trigger, pre- and post-conditions) is equivalent to the requirements. Conflicts between goals may also be captured, and resolved. Trust and security may be modelled by identifying the stakeholders that wish a security goal/requirement, and only assigning agents that are trusted by the stakeholder for the security requirement. Threats may also be modelled by identifying anti-goals of attackers, and guaranteeing that countermeasures are available (not shown in the figure).

Refining TSP requirements into policies can be done in two steps: operationalizing requirements describing a required state transition into logically equivalent operations in the form of <trigger, pre and post condition >. The policy rules could then be defined by translating the <trigger, pre and post condition> operation definition into an < event, condition, and action > rule definition.

4.2 Policies for Reasoning about Trust and Security in Dynamic Virtual Organisation

Compared to classical distributed systems, the scale and dynamicity of VO management pose major challenges in system design and development. This is particularly valid when dealing with trust and security issues. New computational models for trust and security are needed to express emergent and non-functional behaviour.

A VO is viewed as a coalition of geographically dispersed individuals, groups or entire organisations that pool resources to achieve common goals. Resources are virtualised in the way of Grid services. One of the challenges to be tackled is the composition of Grid services taking into account trust and security as well as other non-functional properties such as QoS. By composition we mean either combining two Grid services or creating complex workflows.

Another challenge concerns the management of VOs and its relation to trust and security properties. The formation of VOs must take into account trust-based information such as reputation. Techniques for checking the consistency between general security properties of a VO and those of potential VO participants, as well as techniques to guarantee that the general properties of a VO are respected by all VO members during the whole VO lifecycle.

TSP policies can be used to control dynamic VOs with autonomic properties such as self-organisation, self-adaptation, self-management and self-healing. Another challenge is to investigate how these autonomic properties impact on the trust and security of a VO. Trust can also be seen as an emergent property of Grid systems, and the configuration of a VO can be dynamically adapted to changes in the reputation (or performance) of VO members.

5 Conclusions

This article has (1) reviewed current issues in Grid security and related them to trust, (2) described how a policy-based approach to VO management can enforce trust, security and privacy properties, and (3) suggested how a method to refine high-level trust, security and privacy properties into operational policies. The policies could be used to enforce TSP properties during the different phases of the VO lifecycle. Such a methodology would build on existing methodologies in requirements engineering, and adapt them for the refinement of GRID TSP policies. The TSP policies can be used to enforce TSP properties in two important phases of the VO lifecycle: VO creation and operation. During VO creation local TSP policies have to be matched with global VO policies and some form of negotiation can be required to create the VO. During VO operation, the local TSP policies can be used to generate parameter values when searching, negotiating and using GRID services. The general objective is to automate as much as possible VO management for TSP properties.

6 References

1. T. Grandison, M. Sloman: A Survey of Trust in Internet Applications. IEEE Communications Survey and Tutorials, 3, 2000.
2. A. Josang, R. Ismail, C. Boyd: A Survey of Trust and Reputation Systems for Online Service Provision. To appear in Decision Support Systems, 2005.
3. I. Foster, C Kesselman, S. Tuecke: The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of Supercomputing Applications 15(3), 200-222, 2001.
4. L. Rasmusson, S. Janssen: Simulated Social Control for Secure Internet Commerce. In C. Meadows, editor, Proceedings of the 1996 New Security Paradigms Workshop. ACM, 1996.
5. A.E. Arenas (editor): Survey Material on Trust and Security. CoreGRID Internal Deliverable. 2005.
6. P.J. Broadfoot, A.P. Martin: A Critical Survey of Grid Security Requirements and Technologies. Oxford University Computing Laboratory Technical Report, PRG-RR-03-15, 2003.
7. M. Surridge: A Rough Guide to Grid Security. Technical Report, IT Innovation Centre, V1.1a, 2002.
8. M. Humprey, M.R. Thompson, K.R. Jackson: Security for Grids. In Proceedings of IEEE, 93(3), 2005
9. D. Chadwick: Authorisation in Grid Computing. . Information Security Technical Report, Elsevier, 10(1)33:40, 2005.
10. L. Pearlman, V. Welch, I. Foster, C. Kesselman, S. Tuecke: A Community Authorization Service for Group Collaboration. In Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, 2002.

11. K. Keahey, V. Welch: Fine Grain Authorization for Resource Management in the Grid Environment. In Proceedings of the Third International Woorkshop on Grid Computing, LNCS 2536, 2002
12. M. Thompson, A. Essiari, S. Mudumbai: "Certificate-based Authorization Policy in a PKI Environment," ACM Transactions on Information and System Security (TISSEC), Volume 6, Issue 4 (November 2003) pp: 566- 588
13. Aliferi et al: "VOMS, an Authorization System for Virtual Organization", Across Grids Conference 2003, Springer LNCS 2970, February 2003.
14. A.K. Bandara, E.C. Lupu, J. Moffett, A. Russo; "A goal based approach to policy refinement" Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004
15. Javier Rubio-Loyola, Joan Serrat, Marinos Charalambides, Paris Flegkas, George Pavlou, Alberto Lluch Lafuente: Using Linear Temporal Model Checking for Goal-Oriented Policy Refinement Frameworks. Procedings Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05), June 2005, Stockholm, Sweden.
16. A. van Lamsweerde: *Elaborating Security Requirements by Construction of Intentional Anti-Models* Proceedings of ICSE'04, 26th International Conference on Software Engineering, Edinburgh, May. 2004, ACM-IEEE ,148-157. **Available via ftp anonymous:** //ftp.info.ucl.ac.be/pub/publi/2004/avl-Icse04-AntiGoals.pdf
17. P. Massonet, C. Ponsard: "A Scenario and Goal based Approach for Guaranteeing Quality of Service for Negotiated GRID Service Level Agreements: An Experience Report". Proceedings first International Workshop on Service-Oriented Computing: Consequences for Engineering Requirements (in conjunction with RE05), August 30th, 2005, Paris, France.

Author Index