An XML-Based Framework for Loosely Coupled Applications on Grid Environments

Andreas Hoheisel and Uwe Der

Fraunhofer FIRST, Berlin, Germany
{andreas.hoheisel, uwe.der}@first.fraunhofer.de

Abstract. This paper focuses on an XML-based framework for the description of grid environments and complex grid applications. A so-called Grid Job Handler is used to manage these applications. The GJobDL as part of this framework employs the Petri net concept to define both the workflow and the dataflow of the grid applications. The Grid Job Handler uses these XML representations to control grid applications via the Java Commodity Grid Kit, which interfaces to Globus.

1 Introduction

Grid Computing is a well-known domain in the scientific community with a broad usage [16] [21], but there is a lack of good solutions for engineering applications. This generates a demand for frameworks on top of grid middleware like Globus [7], or UNICORE [20], which fulfill the requirements of today's engineers: easy migration of their applications into the world of grid computing. In this paper, we present an environment for large, geographically distributed and loosely coupled systems. We use the Fraunhofer Resource Grid (FhRG) as a test bed for our framework [8].

One basic service in grid computing environments is resource mapping, which is responsible for finding resources that match specified resource requests. For various kinds of resource requests already exist special solutions like, e.g., the Storage Resource Broker [19], the implementation of Set-Matching algorithms [14], and the Globus Resource Broker in the Globus Toolkit [7]. Our approach is a more general one. To describe the properties and requirements of grid resources, we have developed a generic and extensible family of XML-based languages. In Sect. 2, we present the underlying concepts of the Grid Application Definition Language (GADL) that is used to describe the main classes of grid resources — software, hardware, and data — as well as the composition of resources forming complex grid applications (grid jobs). A grid service termed Grid Job Handler resolves the requirements of these resources at runtime and executes the grid jobs on the most suitable hardware resources.

One widely-used approach to model the workflow of grid jobs are Directed Acyclic Graphs (DAG) [3]. DAGs have a very simple structure and are easy to use, they possess, however, some relevant disadvantages, which we discuss in Sect. 2.2. Because of these limitations we decided to use instead a special type of Petri net to model the workflow and the dataflow between the software components and data resources that build the grid job. Sect. 2.3 gives a brief introduction to Petri nets and their application in grid computing. Sect. 3 describes an implementation of the Grid Job Handler used to process and to control Petri-net-based grid jobs.

2 Grid Application Definition Language

The Grid Application Definition Language (GADL) is a set of XML-based description languages which we have developed in order to assemble and to define complex grid applications on an abstract level with the aim of automatically mapping these applications onto the available hardware and software resources and to control the workflow and dataflow during the execution. The GADL consists of four major parts each covering a different scope:

- **GResourceDL:** The Grid Resource Definition Language is used to describe and to categorize resources. The term "resource" comprises hardware resources (computers, measuring devices, etc.) as well as software components and data. The GResourceDL delegates some special parts of the resource description to the GInterfaceDL and the GDataDL (see below).
- **GInterfaceDL:** The Grid Interface Definition Language is used to describe advanced interfaces of software components that use technologies like CORBA [17] or SOAP [10] for communication. Simple communication techniques via standard IO and files are included in the GResourceDL.
- **GDataDL:** The Grid Data Definition Language provides the description of data that is available as grid resources or data streams that are exchanged between software components.
- **GJobDL:** The Grid Job Definition Language is used to describe grid jobs, i.e. a set of resource descriptions together with the definition of their dependencies and logical interrelations. The description of the participating resources is delegated to the GResourceDL.

Fig. 1 illustrates the role of the GADL within the grid architecture of the FhRG. The GResourceDL descriptions of the grid resources are stored and managed by means of an XML-based repository. We use SQL/XPath queries to retrieve GResourceDL documents or single attributes and elements from the repository. A task-mapping engine maps the given problem onto a set of grid resources that is suitable to solve the problem. Therefore, the user navigates through a task tree in order to constrict the application area of the problem. The description of each resource contains a list of corresponding application areas that are compared with the user requirements. If the application areas of a resource include the required tasks, this resource will be suggested to the user. The result of this interactive task-mapping process is a set of GResourceDL documents containing the description of the basic grid resources involved in the grid job. The graphical Grid Job Builder may be used to assemble and configure the resources to form a coupled grid job that is defined by a GJobDL document.

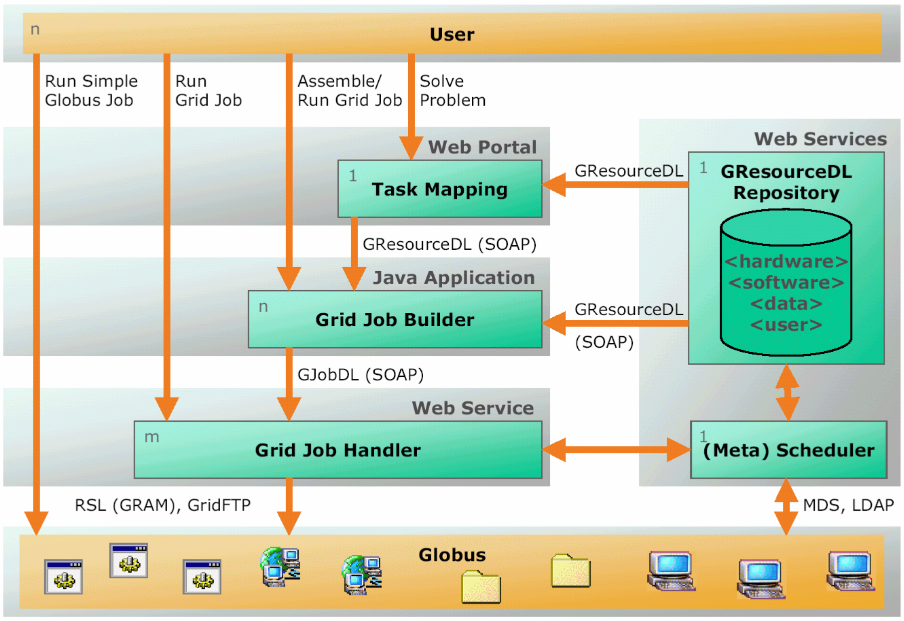

Fig. 1. The grid architecture of the Fraunhofer Resource Grid. The numbers in the upper left corners denote the cardinality

The Grid Job Handler is responsible for the execution of the grid job on a set of suitable hardware resources. Therefore, the Job Handler parses the grid job description, resolves the dependencies between the grid resources, and searches for sets of hardware resources that fulfill the requirements of each software component. A (meta) scheduler is used to select the best-suited hardware resource of each set of matching hardware resources, according to a given scheduling policy (fastest, cheapest, etc.). If using the Globus toolkit [7] as underlying grid middleware, the Job Handler maps the resulting grid resource descriptions to the Globus Resource Specification Language (RSL) [9] and submits the atomic jobs via GRAM [7] to the corresponding grid nodes. This section mainly focuses on the description of grid resources (GResourceDL) and of coupled grid jobs (GJobDL).

2.1 Grid Resource Definition Language

In order to support task-mapping mechanisms for problem-solving environments and resource discovery with regard to the dependencies between grid resources, it is mandatory to have suitable metadata about the grid resources involved in the grid job. Therefore, the Grid Resource Definition Language (GResourceDL) supports the description of six basic resource types:

– concrete software components (type="software"),

- software classes (type="softwareClass"),
- concrete hardware resources (type="hardware"),
- hardware classes (type="hardwareClass"),
- concrete data (type="data"), and
- data classes (type="dataClass").

The distinction between concrete instances of objects on the one hand, and classes of objects on the other hand is very useful when resolving the dependencies between resources during the resource mapping. Here, we use the term *instance* for objects that actually possess a physical (or logical) location represented by a unique location tag, e.g., the IP number of a hardware resource or the URI of a data file. A *class* of objects does not possess a unique location tag, though. A hardware class could be a certain type of CPU (e.g., AMD 1GHz) or the minimal amount of memory (e.g., RAM $>$ 512MB). A software class, for example, could be the operating system "Linux" or the library "glibc6". Other design principles applied to the GResourceDL — besides the distinction between instances and classes — are the concepts of extension and inheritance, thus allowing the recursive formulation of complex and nested resource descriptions with a small set of basic language elements. This is achieved by introducing a language element that allows us to define dependencies between different resource descriptions. Four types of dependencies are supported by the GResourceDL:

- **depends:** This resource depends on or requires other resources, e.g., the software component "lagrange" depends on the software classes "linux", "glibc6", and the hardware class "x86".
- **provides:** This resource provides another resource or contains it. It inherits the properties of the resource provided and may extend it by further properties, e.g., the resource "linux-kernel-2-4-18" provides and extends the software class "linux".
- **conflicts:** This resource must not be used together with other resources.
- **suggests:** It is recommended to use this resource together with other resources.

Example of the XML syntax of the GResourceDL

```
<resource id="lagrange" type="software">
  <dependencies type="depends">
    <resourceRef id="linux" type="softwareClass" />
    <resourceRef id="glibc6" type="softwareClass" />
    <resourceRef id="x86" type="hardwareClass" />
  </dependencies>
</resource>

<resource id="gridNode15" type="hardware">
  <dependencies type="provides">
    <resourceRef id="x86" type="hardwareClass" />
    <resourceRef id="network-ethernet-100" type="hardwareClass" />
    <resourceRef id="linux-kernel-2-4-18" type="softwareClass" />
    <resourceRef id="glibc6" type="softwareClass" />
  </dependencies>
</resource>
```

In the example shown above, the hardware resource "gridNode15" matches the requirements of the software resource "lagrange" if the software class "linux-kernel-2-4-18" provides the resource "linux". The element <resourceRef> is used to define a reference to a resource that is already declared in another place.

The GResourceDL includes two different approaches to declare the properties of the resources. The first approach is provided by including a generic element "parameter". The value of each parameter is defined by the child element "value" and its attribute "op" $\in \{=, >, \geq, <, \leq\}$. The name of the parameter is declared by its attribute "name". This generic approach has the disadvantage that the names and types of the parameters cannot be specified in advance and typing errors may consequently result in runtime errors. Another shortcoming is the overhead needed to categorize and search specific properties. Therefore, we include a second approach in the GResourceDL, declaring frequently-used properties by own element tags that can be checked directly by means of a validating XML parser. Frequently-used resource properties that are needed in a grid environment are, e.g., the resource location, the execution location, the input and output ports, as well as accounting and authorization information.

2.2 Grid Job Definition Language

We use the Grid Job Definition Language (GJobDL) to describe the dynamic behavior of complex grid applications that may consist of several coupled software components as well as their input and output data. This meta data is needed to map the grid application onto the underlying grid middleware and to control the workflow and dataflow of the grid application. The GJobDL description of a grid job contains the GResourceDL description of the basic resources that are required to define the grid job and the model of the grid job workflow using the concept of Petri nets [18]. In many other grid projects, the workflow of coupled grid applications is modeled using Directed Acyclic Graphs (DAG) (see Fig. 2). One example for this approach is UNICORE, where so-called AbstractJobs are defined on the basis of DAGs [20]. Other prominent grid projects using DAGs are Condor [3], GriPhyN [6], Cactus [4], and Symphony [15]. DAGs are widely spread due to their simple structure, they possess, however, some relevant disadvantages: As DAGs are directed, it is impossible to define bidirectional coupling schemes between software components. DAGs are acyclic, so it is not feasible to define loops (while ... do). A DAG only describes the behavior, but not the state of the system. DAGs generally describe only the workflow, and not the dataflow. Because of these limitations we decided to use Petri nets (see Fig. 3) instead of DAGs. The idea to use Petri nets to control the workflow of complex applications has been borrowed from the Graphical Simulation Builder that is being developed by the Potsdam Institute for Climate Impact Research (C. Ionescu, pers. comm.).

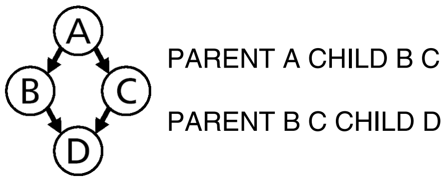

Fig. 2. Example of a Directed Acyclic Graph (DAG)

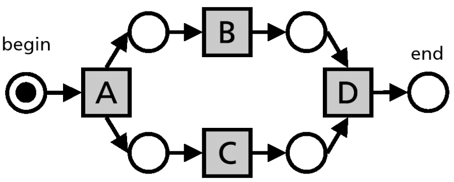

Fig. 3. Example of a Petri net equivalent to the Directed Acyclic Graph in Fig. 2

2.3 Petri Nets

Petri nets are graphical representations of the workflow of discrete systems. In contrast to DAGs, which only describe the dynamical behavior of the system, Petri nets also describe the system state [18]. The type of Petri nets we introduce here corresponds to the concept of Petri nets with individual tokens (colored Petri net) and constant arc expressions which are composed of places, denoted by circles (◯), transitions, denoted by boxes (□), arcs from places to transitions (◯ → □), arcs from transitions to places (□ → ◯), individual and distinguishable objects that flow through the net as tokens, an initial marking that defines the objects which each place contains at the beginning, and an expression for every arc that denotes an individual object. A place p is called input place (output place) of transition t if an arc from s to t (from t to s) exists. A brief introduction to the theoretical aspects of colored Petri nets can be found, e.g., in [11].

We use Petri nets not only to *model* the workflow, but furthermore to *control* the workflow of complex grid applications. In most cases, the workflow within grid applications is equivalent to the dataflow, i.e., the decision when to execute a software component is taken by means of availability of the input data. Therefore, the tokens of the Petri net represent real data that is exchanged between the software components in the grid. In this case, we use Petri nets to model the interaction between software resources represented by software transitions, and data resources represented by data places. In some cases, however, the workflow is independent from the dataflow, and in addition to the data places and software transitions we have to introduce control places and control transitions. The corresponding tokens contain the state of the process (e.g., done, failed). Therefore, we extend the classic definition of Petri nets by adding some new properties: Places can either be data places or control places. Data places are references to resources of type "data". Control places may contain tokens that include information about the process state. The capacity of each place is one, i.e., each place can either contain none or one token. Transitions can either be software transitions or control transitions. Software transitions are references to resources of type "software" and may possess several input and output ports. Control transitions evaluate logical conditions. Every arc either begins at a place and ends at a transition (or a specified port of a software transition) or begins at a transition (or a specified port of a software transition) and ends at a place.

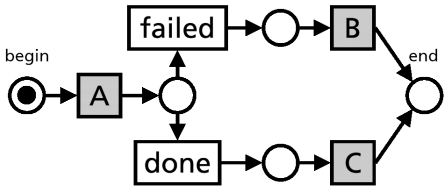

Fig. 4. Example of a Petri net that represents the condition: if (A = done) then C else B

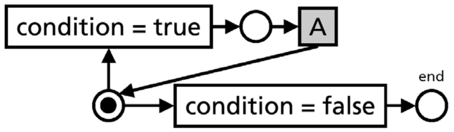

Fig. 5. Example of a Petri net that represents the loop: while (condition = true) do A

Objects that are stored as tokens on data places represent real data. The port type of a transition defines the expression of the arc that is connected with this port.

If an arc connects a data place with a port of a software transition, the software component that is represented by the software transition may only be executed if the input data is available, i.e., the corresponding input place must contain a token. The content of the data place, i.e. the data content of the resource that is linked to the place, is transferred to the corresponding port of the software resource.

Petri nets possess special characteristics that can be defined mathematically and are used to analyze and classify Petri nets. Terms like, e.g., conflict, confusion, contact, trap and deadlock are well-defined properties of Petri nets that may be helpful when optimizing the workflow of a grid job. When using Petri nets, it is easy to model all kinds of discrete workflows by means of the three basic components — places, transitions, and arcs. The actual state of the workflow is represented by the marking of the Petri net. A good overview over the different workflow models using Petri nets can be found in [1]. Each transition represents a simple task or atomic job that may be replaced by a whole Petri subnet, according to the refinement paradigm. Several of these tasks may be executed sequentially (e.g., transitions A, B and D in Fig. 2) or in parallel (e.g., transitions B and C in Fig. 2). It is also possible to define conditions (Fig. 4) or loops (Fig. 5). Fig. 6 shows an example of the graphical representation of a whole grid application. An extra control flow may be modeled additionally to the data flow in order to include a simple kind of fault management (not shown here).

There already exist several approaches to describe Petri nets with XML-based description languages, e.g., the Petri Net Markup Language (PNML) [12]. These approaches, however, do not comply with the requirements for the use as a grid job definition language, as mechanisms to connect the transitions and places with real grid resources are missing. As a consequence, we included a proprietary XML syntax — similar to the PNML — in the GJobDL. The job description consists of the declaration of the places, transitions, and arcs that build the Petri net of the grid job. Transitions and places may be linked to external or internal GResourceDL descriptions. Control transitions may possess conditions that are

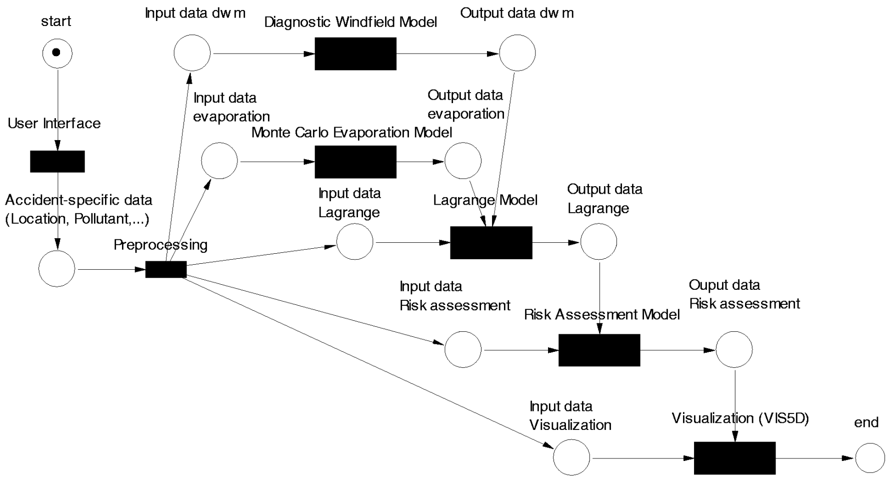

Fig. 6. Example of a Petri net representing a coupled grid application that is part of the *Environmental Risk Analysis and Management System* (ERAMAS), which is a simulation-based accident management system for environmental risks caused by dangerous substances

evaluated prior to the firing of enabled transitions. Places may have an initial marking that defines the initial state of the grid job.

3 Grid Job Handler

With the GJobDL document of a grid application, we possess an adequate description of the involved resources as well as a model of its workflow and dataflow. Now we need a driver or handler that parses this grid job description and executes all the single components of the grid job in the right order, at the right time, and on the right place within the grid environment. Here we present a prototype of such a grid job handler on the basis of the GADL that we have developed within the context of the FhRG and that supports basic features like automatic file transfer, secure connections, monitoring of grid jobs, etc. For the communication between the Job Handler and the Globus grid middleware, a patched version of the Java Commodity Grid Kit [13] is used. The current implementation of the Job Handler supports only software components that can be described as black boxes with well-defined input and output ports. If the software component is executed it reads input data through the input ports and writes output data through the output ports. In fact, these requirements may constitute a limitation for potential grid applications, e.g., for software components that use streaming technologies. Here, further work has to be done to establish a more general component environment, as it is described, e.g., by the Common

Component Architecture (CCA) [2]. The following steps are iteratively invoked in the kernel of the Job Handler:

1. **Get all enabled transitions** of the grid job
2. **Evaluate the conditions** of the enabled transitions
3. If necessary, **invoke** special **method calls** that are referred by the transition (transfer executable, transfer data, unpack, ...)
4. If the transition references a software component, **invoke the resource mapping** in order to get a set of matching hardware resources
5. **Ask the scheduler** for the best-suited hardware resource to execute the software component out of the set of matching hardware resources
6. **Submit the software component** to the hardware resource, e.g., using the GRAM protocol
7. **If the software component exits** (status is failed or done), let the corresponding **transition fire** (remove tokens from input and put tokens to the output places)
8. **Repeat 1-7**, until there are no more enabled transitions left

Note that it does not matter how complex the grid application becomes, the kernel of the Job Handler remains the same for every type of workflow. The Job Handler can be used locally or be accessed remotely by using Web Service [5] on basis of SOAP [10]. Also, a client Java-API for the connection to the Job Handler Web Service is available. The Job Handler can be registered as Web Service in a UDDI registry, enabling the remote job submission via a web browser.

4 Conclusions and Future Work

Computing grids give us an execution environment for distributed applications with considerable advantages over homogeneous computing farms or single computer systems. However, in most cases it is much more complex to write grid-enabled applications. Here, we have presented a generic framework to build complex and loosely coupled applications and to execute them on the grid. The Petri-net-based approach enables the easy aggregation of complex workflows, including conditions and loops, by using only few basic elements. This framework should adapt to the service-oriented architecture of Globus 3.0-based grids (OGSA).

For pragmatic reasons, we use a centralized repository to store and discover the grid resource descriptions, which is contrary to a decentralized grid concept. Key issues, where further work has to be done, are the scalability of the framework, software and hardware benchmarks for advanced scheduling mechanisms and the semi-automatic generation of resource descriptions. We plan to validate our solution with more applications in the engineering domain.

References

1. van der Aalst, W. M. P.: The Application of Petri Nets to Workflow Management. The Journal of Circuits, Systems and Computers **8** (1998) 21–66

2. Armstrong, R., Gannon, D., Geist A., Keahey K., Kohn, S., McInnes L., Parker, S., Smolinski, B.: Toward a Common Component Architecture for High-Performance Scientific Computing. In: Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC9). Redondo Beach, California (1999)
3. Basney, J., Livny, M.: Deploying a High Throughput Computing Cluster. In: High Performance Cluster Computing, Vol. 1. Prentice Hall PTR (1999)
4. The Cactus Project: Cactus 4.0 Users' Guide. `http://www.cactuscode.org` (2002)
5. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language. W3C, `http://www.w3c.org/TR/wsdl` (2001)
6. Deelman, E., Kesselman, C., Mehta, G., Meshkat, L., Pearlman, L., Blackburn, K., Ehrens, P., Lazzarini, A., Williams, R., Koranda, S.: GriPhyN and LIGO, building a virtual data grid for gravitational wave scientists. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC11). Edinburgh, Scotland (2002) 225–234
7. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers, Inc. (1999)
8. Fraunhofer Resource Grid. `http://www.fhrg.fhg.de` (2003)
9. The Globus Project: The Globus Resource Specification Language RSL v.1.0. `http://www.globus.org/gram/rsl_spec1.html` (2000)
10. Gudgin, M., Hadley, M., Moreau, J.-J., Nielsen, H. F.: SOAP version 1.2. W3C `http://www.w3.org/TR/2001/WD-soap12-20010709` (2001)
11. Jensen, K.: An Introduction to the Theoretical Aspects of Coloured Petri Nets. Lecture Notes in Computer Science, Vol. 803. Springer-Verlag, Berlin Heidelberg New York (1994) 230–272
12. Jüngel, M. Kindler, E., Weber, E.: The Petri Net Markup Language. In: Philippi, S. (ed.): 7. Workshop Algorithmen und Werkzeuge für Petrinetze. Universität Koblenz-Landau (2000) 47–52
13. von Laszewski, G., Foster, I., Gawor, J., Lane, P.: A Java Commodity Grid Kit. Concurrency and Computation: Practice and Experience **13** (2001) 643–662
14. Liu, C., Yang, L., Foster, I., Angulo, D.: Design and Evaluation of a Resource Selection Framework for Grid Applications. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC11). Edinburgh, Scotland (2002) 63–72
15. Lorch, M., Kafura, D.: Symphony — A Java-based Composition and Manipulation Framework for Computational Grids. In: Proceedings of the CCGrid2002. Berlin, Germany (2002)
16. Nasa Information Power Grid. `http://www.ipg.nasa.gov` (2003)
17. Object Management Group: The Common Object Request Broker: Architecture and Specification. `http://www.omg.org/cgi-bin/doc?formal/01-02-33` (2001)
18. Petri, C. A.: Kommunikation mit Automaten. Ph.D. dissertation. Institut für Instrumentelle Mathematik, Bonn (1962)
19. Rajasekar, A., Wan, M., Moore, R.: MySRB and SRB — Components of a Data Grid. In: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC11). Edinburgh, Scotland (2002) 301–310
20. Romberg, M.: The UNICORE Architecture: Seamless Access to Distributed Resources. In: Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8). Los Alamitos, CA (1999) 287–293
21. Segal, B.: Grid Computing: The European Data Project. In: IEEE Nuclear Science Symposium and Medical Imaging Conference. Lyon, France (2000)