Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

# Invocation of operations from script-based Grid applications

Maciej Malawski[a,*], Tomasz Bartyński[b], Marian Bubak[a,c]

[a] *Institute of Computer Science, AGH, Mickiewicza 30, 30-059 Kraków, Poland*

[b] *Academic Computer Centre CYFRONET AGH, Nawojki 11, 30-950 Kraków, Poland*

[c] *Informatics Institute, Universiteit van Amsterdam, Kruislaan 403, 1098 SJ Amsterdam, The Netherlands*

ARTICLE INFO

*Article history:*
Received 20 October 2008
Received in revised form
30 March 2009
Accepted 13 May 2009
Available online 27 May 2009

*Keywords:*
e-science application
Application building
Grid computing
Virtual laboratory
EGEE
DEISA

ABSTRACT

In this paper we address the complexity of building and running modern scientific applications on various Grid systems with heterogeneous middleware. As a solution we have proposed the Grid Operation Invoker (GOI) which offers an object-oriented method invocation semantics for interacting with diverse computational services. GOI forms the core of the ViroLab virtual laboratory and it is used to invoke operations from within *in-silico experiments* described using a scripting notation. We describe the details of GOI (including architecture, technology adapters and asynchronous invocations) focusing on a mechanism which allows adding high-level support for batch job processing middleware, e.g. EGEE LCG/gLite. As an example, we present the NAMD molecular dynamics program, deployed on EGEE infrastructure. The main achievement is the creation of the Grid Object abstraction, which can be used to represent and access such diverse technologies as Web Services, distributed components and job processing systems. Such an application model, based on high-level scripting, is an interesting alternative to graphical workflow-based tools.

## 1. Introduction

Modern researchers, mostly in natural and life sciences, solve highly complex problems with so-called *in-silico* experiments. These high-level applications require large computational power and storage and may combine various software tools or software deployed on heterogeneous distributed resources. Experiments need to employ both legacy tools, usually run as jobs on Grid infrastructures, and software implemented and exposed to modern middleware technologies, such as Web Services or components. This, however, implies a problem when interfacing a software that not only relies on different middleware packages, but also on diverse interaction models [1].

Grid infrastructures have been considered the most appropriate platform for computational science for many years [2] and, consequently, many European projects providing such production infrastructures have been created, including EGEE [3] and DEISA [4]. Their main goal is to provide production infrastructure for high-throughput and high-performance computing respectively. In addition, there are also other initiatives, which focus on various middleware frameworks, often based on service-oriented architectures or component models. Their aim is to provide computational resources virtualized at a higher level, e.g. in the form of Web Services. The service-oriented approach offers access to software using well-defined interfaces (the Web Services from the European Bioinformatics Institute [5] are a good example), while production infrastructures provide relatively low-level interfaces to computing resources, often limited to simple batch job submission. Building applications that use these infrastructures remains a challenging task, due to the heterogeneity of Grid middleware and different programming models. Therefore, the research concerning tools for the development of such programs is of great importance.

Such a challenge is faced by *virtual laboratory*, which is a set of tools that form a collaborative and distributed space for *in-silico* experiments. This environment supports scientists in developing, sharing and executing experiments. An example of a virtual laboratory is a platform being developed in the scope of the *ViroLab* [6] project. Experiments in this virtual laboratory are high-level applications which orchestrate many computational tasks running on the Grid. The notation used for specifying experiment plans uses the Ruby scripting language [7]. This approach allows specifying arbitrary complex experiments in a modern object-oriented dynamic language, thus giving the programmer full control and flexibility in the area of experiment design. Scripts, written in a full-fledged programming language, can define experiment logic using a rich set of control structures and also perform some computations locally. Scripts are particularly convenient when there is a need to combine high-level control structures of an application with some *glue* code necessary to, e.g. convert output of one service to the format required by another one, or to perform some simple

[*] Corresponding author. Tel.: +48 126174466; fax: +48 126339406.
*E-mail addresses:* malawski@agh.edu.pl (M. Malawski), t.bartynski@cyfronet.pl (T. Bartyński), bubak@agh.edu.pl (M. Bubak).

0167-739X/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2009.05.012

local processing which does not have to be delegated to an external service. Our experience with the virtual laboratory indicates [8,9] that such an approach is an interesting and convenient alternative to many existing scientific workflow systems which use graphical notation [10–12].

The main research problem which we focus on in this paper is how to access the underlying Grid resources from such high-level applications. Solving this requires development of proper abstractions, which can remain simple and intuitive to use as well as covering a wide range of middleware types: service-oriented, component-based or using job processing model. As a result of our investigations, a dedicated module of the virtual laboratory, called the Grid Operation Invoker (GOI) [13], has been developed. It applies an object-oriented model with remote procedure call semantics to dispatch computation in a uniform manner using diverse middleware technologies. During the first development stage we provided support for Web Services and MOCCA [14] component technologies. MOCCA is a CCA-compliant [15] framework for building and running applications on the Grid. Advantages of the component-based approach include the possibility of deployment of custom-developed software modules on the available infrastructures, as well as more flexible constructing of applications by connection component ports. To allow users to interact with various middleware systems, GOI introduces multiple levels of abstractions, called *Grid Objects*.

In this paper we describe in detail the structure of the Grid Operation Invoker and how it supports middleware technologies which are based on the job processing model implemented in EGEE and DEISA Grid infrastructures. Such projects provide scientists with computational power, storage and a wide range of scientific applications; yet it should be noted that their resources are accessed with tools dedicated for one specific middleware package, which enables submitting jobs or sequences of jobs. In the case of ViroLab, we have to deal with gLite [16] which is installed on EGEE and also with the Application Hosting Environment (AHE) [17] which is a lightweight middleware focused on accessing applications on the Grid in a user-friendly way and can provide interface to DEISA as well. In order to solve a scientific problem in a virtual laboratory, it is often required to combine results produced by a set of these tools, as well as by local applications. This procedure is time-consuming and can be performed only by skillful users. Research can be facilitated by integrating all local tools, Web Services and Grid jobs into a single experiment which uses a uniform and simple notation to describe all steps of a scientific process and automate it entirely. In this paper, we also describe how this can be achieved using the proposed Grid Operation Invoker.

This paper is organized as follows: Section 2 provides an overview of the related work on providing access to Grid middleware systems. Subsequently, in Section 3, we introduce the main concepts of the Grid Operation Invoker and then, in Section 4, its role in the virtual laboratory. In Sections 5 and 6, a detailed description of enhancements provided to add support for job-based middleware systems is presented on the example of LCG/gLite (EGEE). Section 7 describes support for asynchronous (non-blocking) invocation of operations. In Section 8 we report on experiments which were performed in the virtual laboratory exploiting GOI. The final section includes a summary and a brief presentation of future work. Our preliminary approach to running script-based applications on EGEE Grid was presented in [18].

## 2. Related work

Numerous software frameworks have been developed to provide high-level access to Grid services using heterogeneous middleware systems. The Grid Application Toolkit (GAT) [19], currently evolving into the Simple API for Grid Applications (SAGA), provides a language-neutral API to basic Grid use cases, such as operations on files, monitoring events, resources, jobs, information exchange, error handling and security. However, it does not introduce an object-oriented API to invoke applications. A similar approach has been undertaken by the authors of the Grid Services Base Library (GSBL) [20]; however it is still limited to such operations as job submission and file transfers. Multiple Grid and cloud computing middleware systems can be also accessed using g-Eclipse tools [21], but they do not support high-level application-oriented interfaces.

Another high-level approach is implemented in NetSolve/GridSolve [22], which is an RPC-based system where a client delegates the execution of an operation to a selected server providing input parameters. The server executes the appropriate service and returns output parameters or error status to the client. Since GridSolve requires installation of specific servers, its usage on such infrastructures as EGEE is not straightforward.

Portal-based systems, like GridPortlets and OGCE [23], also provide a means for accessing multiple middleware technologies. These solutions are usually dependent on a specific portal technology (e.g. Java portlets), although recently (in the VINE toolkit [24]) there have been attempts to extend their usability to more generic applications.

One should also consider systems used for migrating so-called *legacy code* applications to Grid or to Grid Services. Examples of such systems include LGF [25] which wraps legacy code as Globus 4 services on a fine-grained level, or GEMLCA [26], which offers a more coarse-grained approach. However, they are limited to a single middleware suite, such as Globus 4.

Other platforms which aim to facilitate the usage of Grids by scientific applications include workflow systems [27], such as K-Wf Grid [11], which manages workflows on multiple levels of abstraction; Kepler [10], which allows integrating multiple actor models, and Taverna [12], successfully applied to many life-science applications. The main drawback of workflow systems, in comparison to the scripting approach, is the limited expressiveness of graphical notations when applied to more complex experiments.

None of these approaches propose a complete solution for running applications on different Grid middleware systems.

## 3. Goals and concepts of Grid Operation Invoker

The Grid Operation Invoker is designed as a module of the Virtual laboratory engine and it is responsible for communication with diverse underlying middleware technologies. Having analyzed the needs of the scientific community, as well as similar solutions, we have defined requirements for the Grid Operation Invoker system. The main functional requirements are as follows:

- to provide uniform and coherent interface to the functionality of applications accessible using heterogeneous middleware,
- to provide APIs on both high and low level of abstraction (allowing developers to define the required functionality or choose a specific instance),
- to handle required data conversions (from Ruby types to SOAP and Java objects) in a transparent manner.

Besides listed functional features, the GOI module should also conform to non-functional requirements, including the following:

- to integrate all external libraries,
- to remain OS-independent,
- to ensure ease of extending the system with support for emerging middleware technologies,
- to enable operation both as a standalone solution and as part of a bigger system, such as a virtual laboratory,
- to remain unobtrusive at the server (provider) side.

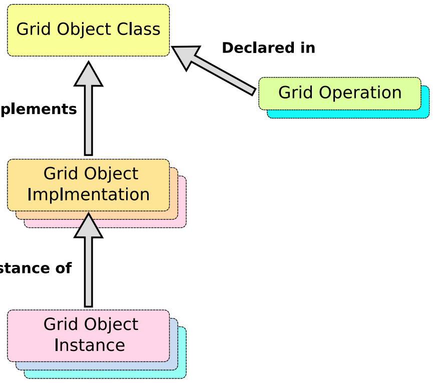

**Fig. 1.** Three levels of the abstraction over the Grid environment: Grid Object class, implementation and instance.

In order to fulfill these goals we introduced the concept of *Grid Objects* which are representatives of services, components or jobs on the *client side*. Grid Objects are instantiated within the experiment script and by invoking methods on them a programmer is able to access specific operations on remote resources. It is noteworthy that GOI is focused on access to *computing* middleware and resources. Access to data resources remains out of scope of GOI, and in ViroLab virtual laboratory this complementary aspect is handled by a dedicated tool, i.e. the Data Access Client [28].

Fig. 1 illustrates the Grid Object hierarchy. The main reason behind introducing this hierarchy and its associated layers of abstraction was that the complexity of the heterogeneous, distributed environment should be hidden from end users. Developers of an application should not be concerned with manually interfacing all underlying middleware technologies — they should instead be focused on the problem they are solving.

Each *Grid Object Class* is an abstract entity which defines a set of *Grid Operations*. These operations are invoked from the script, while the actual computation is performed on a remote machine. Each Grid Object class may have multiple *Implementations* with different middleware technologies representing the same functionality. Each of the implementations may have multiple *Instances*, possibly running on different resources and thus with different levels of performance. Grid Object instances of a specific class may use a variety of middleware suites and therefore must be interfaced using their specific protocols. Moreover, Grid Objects may have various properties, such as stateless or stateful interaction mode, synchronous or asynchronous operation invocation etc. Furthermore, Grid Objects may be private (for instance, the user deploys a component in his/her experiment and only he/she can access it) or shared between experiment runs and between users (for example, publicly available services). All these properties are part of technology information that is used while creating a Grid Object representative. Developers are not concerned with finding the optimal instance and interfacing it; however, they must be aware of the properties of each Grid Object. For instance, they must know whether the Grid Object they are using preserves state between invocations of operations.

A sample script demonstrating the invocation of the Decision Support System (DSS) which suggests a drug ranking for a patient with a specific set of HIV mutations is shown in Fig. 2. `GObj` is a factory for Grid Objects: in line 4 it is used to create an instance representing the DSS Web Service. Upon instantiation, the operations of a Grid Object can be invoked directly, as seen in line 6. A usage of Ruby string operations, such as `split()` in line 5, enables simple conversions, which would be nontrivial in the case of graphical workflow systems and would often require specific converter or adapter services.

```
1   require 'cyfronet/gridspace/goi/core/g_obj'
2
3   begin
4     dss = GObj.create('org.virolab.DrugRankingSystem2')
5     mutations = 'P1M I2L S3T P4Q E6G T7C'.split(' ')
6     res = dss.drs('ANRS',
7                   'reverse_transcriptase',
8                   mutations)
9     puts res
10  end
```

**Fig. 2.** A sample ViroLab experiment invoking the drug ranking Web Service using the Grid Object library.

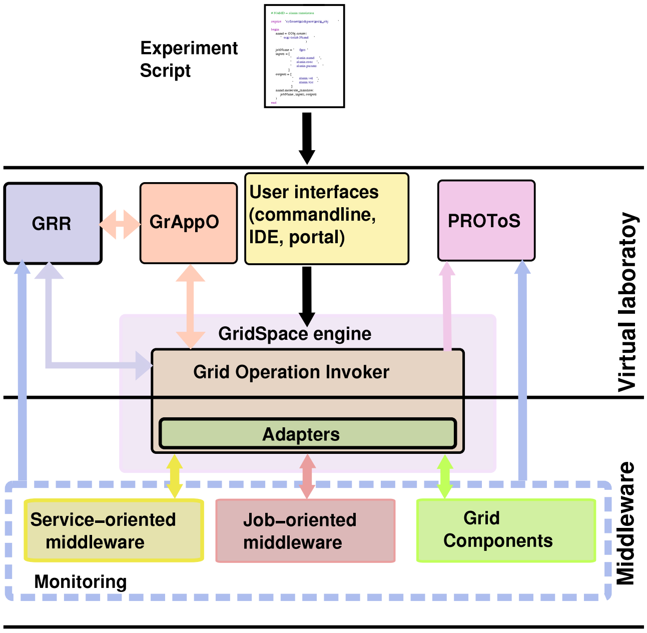

**Fig. 3.** Grid Operation Invoker in the context of the GSEngine.

## 4. Architecture of Grid Operation Invoker

The Grid Operation Invoker is a library that provides a uniform interface to multiple middleware technologies. It supports abstraction over the heterogeneous environment as described in Section 3.

Fig. 3 shows how GOI is positioned in the context of other modules of the virtual laboratory. GOI is a part of GridSpace Engine (GSEngine), which is the main execution server for experiments, with an embedded JRuby interpreter. Descriptions of technical information of Grid Objects are stored in the external Grid Resource Registry (GRR) service and the Optimizer module (GrAppO) is responsible for selection of optimal instances if more than one instance is available for a specific object. The optimizer plays a role similar to a broker and a scheduler known in workflow systems and it uses resource information from the monitoring subsystem. GOI also publishes events to the Provenance Tracking System (PROToS) [29] which stores all the historical execution data for the purpose of experiment result analysis and possible future validation.

The Grid Operation Invoker has a modular architecture and all components have well-defined interfaces. As a result, code reusability, ease of extending the system and interoperability with external components are ensured. For instance, if it is required to cooperate with another optimizer, this can be easily accomplished by providing a client implementing the required interface and specifying that GOI should use this class. The same pattern applies to using a registry.

**Fig. 4.** Structure of the Grid Operation Invoker.

The Grid Operation Invoker system is divided into three packages (see Fig. 4). The `core` package contains main parts of the system. `GObj` is a factory that provides the uniform interface for creating representatives for `Grid Objects`. It implements the algorithm of choosing resource, loading an adapter for the appropriate technology and producing a representative. It uses dedicated clients to communicate with the optimizer and the registry, which delegate queries to external systems, such as GrAppO and the Grid Resource Registry. `Adapter` class is capable of producing a representative (an object of the `Resource` class) for a Grid Object using one specific middleware technology. All adapter and resource classes are included in the `adapters` package. Finally, the `utils` package consists of any additional classes that are used by the GOI. Good examples are `Future` class, which enables asynchronous operation invocations (see Section 7), `JobSpec` and `GliteWmsUIWrapper` classes that are used by the gLite adapter.

GOI is implemented in JRuby, taking advantage of the dynamic features provided by the language, such as dynamic method dispatch and code generation and evaluation at run-time. This enables GOI to adapt to the heterogeneous and dynamic nature of the computational environment. During interpretation of the script, at the line in which the `GObj` factory is called to produce a representative for a `Grid Object`, the GOI library comes into action. An optimizer is queried for an optimal resource of the requested `Grid Object class` and returns a unique `id` of the selected resource. Based on this `id`, the technology information which describes the resource in technical terms (such as communication protocols, endpoints etc.) is retrieved from the resource registry. Once this technical data is known, a dedicated adapter for a specific middleware technology is loaded and a representative is created and used in the script as an ordinary Ruby object. For Web Services it is enough to know the WSDL description, or the names of methods, and the SOAP endpoint (if no WSDL is available). Upon invocation of a method on the Grid Object representative, the resource object uses dynamic method dispatch to delegate the call using the protocol supported by the specific middleware technology. In the case of SOAP-based services a standard Ruby library is used, while e.g. in the case of MOCCA the adapter uses Java client API to invoke remote methods on components. Ruby language employs the *duck typing* paradigm which assumes that types of objects are not checked, but it is checked if an object responds to a specific method with a specific number of arguments. This approach is very convenient, although calling a remote Grid Operation just to find that the instance does not provide such an operation may prove too expensive. Therefore, GOI checks the technology information to determine whether a given Grid Operation is available for a Grid Object representative. If not, an error is reported. What is more, GOI catches all exceptions in remote Grid Operations in order to handle them or report them in a user-friendly manner.

Whenever required, the developer can bypass the optimizer by using a low-level API, but in such a case, a unique id or technology data needs to be provided.

As already mentioned, the GOI can be easily extended. In order to add support for other types of middleware, it is required to implement an adapter class and a resource class. Developers implementing support for external middleware packages may use a wide range of libraries. These include standard Ruby libraries, JRuby gems as well as Java libraries, which can be imported and used within JRuby scripts. What is more, the scripting nature of JRuby facilitates wrapping command-line tools, such as gLite WMS User Interface.

A Web Services adapter has been available from the beginning, implemented using the SOAP package from the standard Ruby library. MOCCA components are also supported and the adapter was implemented using the Java-based MOCCA client library which supports dynamic method invocation. Adapters for job-based middleware technologies such as gLite and AHE are the subject of recent research and are described in detail in the following sections.