Future Generation Computer Systems 26 (2010) 138–146

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.

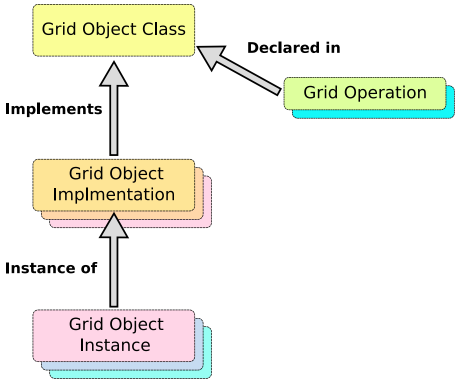

Fig. 1. Three levels of the abstraction over the Grid environment: Grid Object class, implementation and instance.

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.

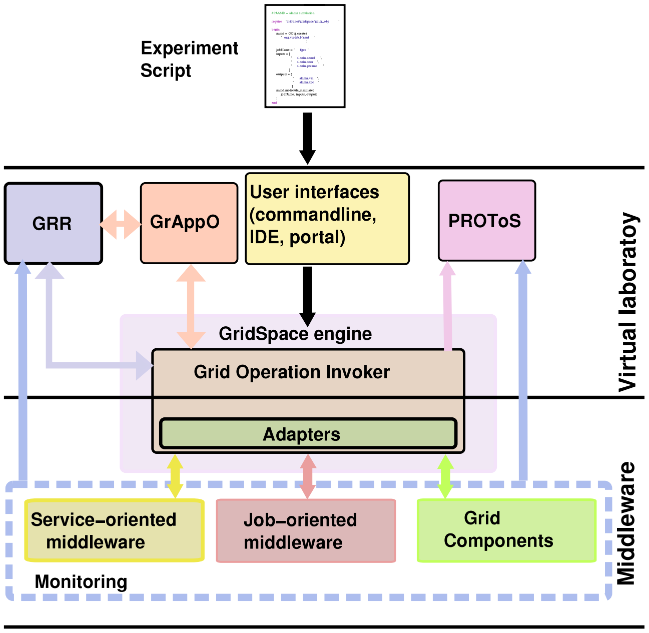

Fig. 3. Grid Operation Invoker in the context of the GSEngine.

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.

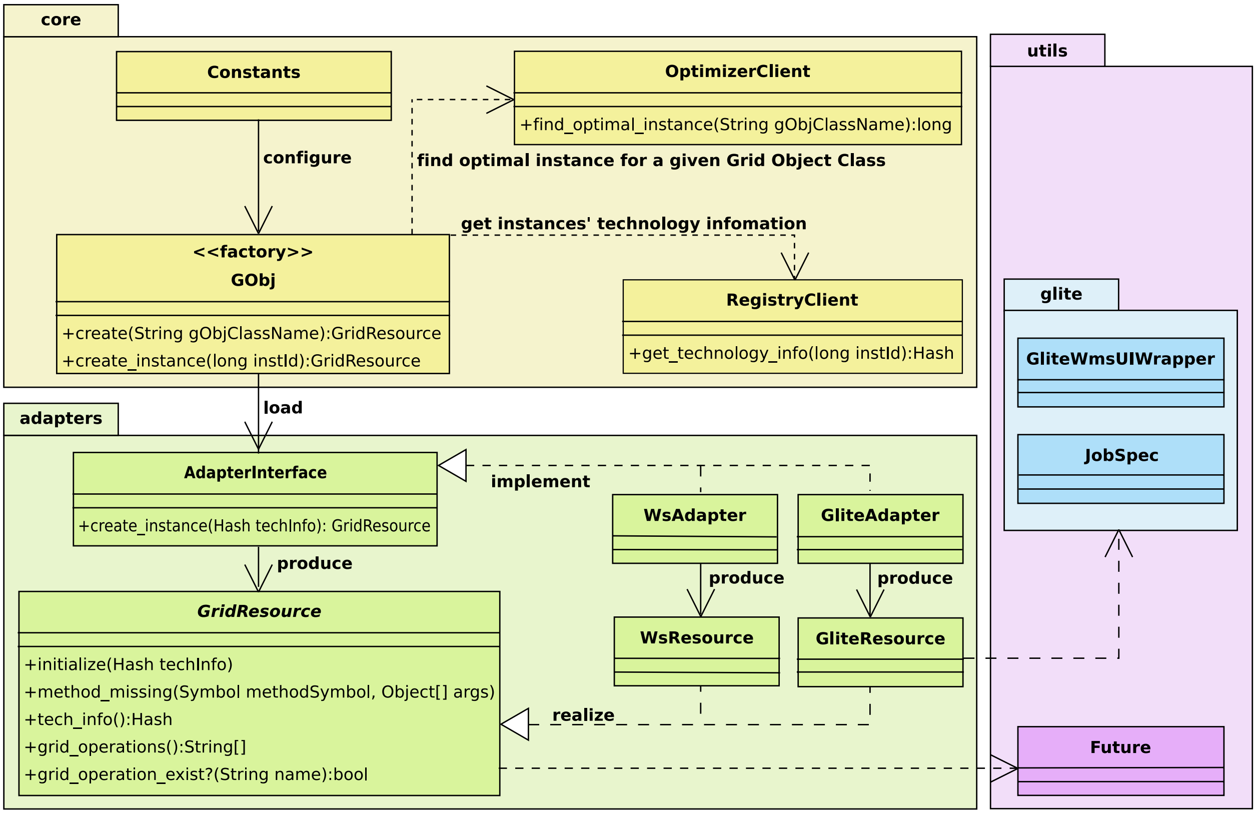
core
Constants
OptimizerClient
+find_optimal_instance(String gObjClassName):long
configure
find optimal instance for a given Grid Object Class
get instances' technology infomation
<<factory>>
GObj
+create(String gObjClassName):GridResource
+create_instance(long instId):GridResource
RegistryClient
+get_technology_info(long instId):Hash
load
adapters
AdapterInterface
+create_instance(Hash techInfo): GridResource
implement
WsAdapter
GliteAdapter
produce
produce
produce
GridResource
+initialize(Hash techInfo)
+method_missing(Symbol methodSymbol, Object[] args)
+tech_info():Hash
+grid_operations():String[]
+grid_operation_exist?(String name):bool
WsResource
GliteResource
realize
utils
glite
GliteWmsUIWrapper
JobSpec
Future

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.

5. Extending GOI with local gems

Grid Objects which represent application-specific functionality are often referred to as *gems*, by analogy to RubyGems [7] — a standard for distributing Ruby libraries. Examples of *ViroLab gems* are such services as the Drug Resistance Service [30] or the RegaDB HIV sequence alignment and subtyping tools [31], all wrapped as Grid Objects.

In addition to the Grid Objects corresponding to remote computations, *Local gems* are introduced as a way of representing local computation as a Grid Object. From the application developer's point of view local gems are another computational technology and are accessed via the same uniform interface as other technologies. Local gems enable the developer to download the source of a Ruby class, evaluate the source at run-time and execute it locally. They facilitate sharing single classes that provide functionality usable for a scientific community, which is, however, too lightweight to be deployed as a Web Service. Local gems are registered in the Grid Resource Registry as a *Grid Object Instance* and their source code is stored in the registry.

It is interesting to compare local gems to experiment scripts. Both are written in Ruby and can provide some interesting processing functionality. Experiments are stored in an application repository [8] which is based on the SVN version control system and can be shared between virtual laboratory users. Local gems, on the other hand, are stored in the Registry. Experiments are intended to be executed by end users (scientists or medical doctors in ViroLab), while local gems are smaller building blocks, to be used by experiment developers. For this reason, we decided that both forms of sharing application scripts in the virtual laboratory are useful and complementary. Additionally, it is possible to register interesting experiments, or parts thereof, as local gems, to encourage code reuse.

6. Support for job-based middleware

In order to integrate job-oriented middleware such as the EGEE LCG/gLite with the Grid Operation Invoker, an object-oriented representative of a job is required. According to the architecture (Section 4) this requires a technology adapter (*Grid Object implementation*) which would delegate the invocation of its operations to the submission of jobs using specific Grid middleware, and return a result upon successful completion of a job.

In contrast to the already implemented adapter classes, capable of producing client-side *Grid Object instance* representatives of Web Service and MOCCA middleware, it is not possible to implement a generic factory for representatives of jobs. Web Service and MOCCA components are, by their nature, object-oriented and are contacted using a well-defined interface. Representatives of *Grid Object instances* published with these technologies are actually stubs (proxies) which provide the same interface and therefore can be generated automatically. On the other hand, Job-oriented middleware enables us to execute command-line applications which do not provide a remote API. Functionality provided by an application is organized in a set of methods, and is determined on the basis of command-line input parameters. As a consequence of this fact, the application has to be wrapped with a special class that exposes its functionality as Grid Object methods.

Each wrapper class should be application-specific. It is common practice in various legacy-code wrapping systems to define a special descriptor language (e.g. XML-based) to specify the mapping between object operations and specific command-line parameters or program execution. In our case, since Ruby is used as the implementation language, it is natural to also use Ruby for specification of this mapping. Therefore, we have decided that wrapper classes would be local gems, able to prepare inputs, submit a job, manage it and retrieve results. Such local gems, in turn, can use a lower-level Ruby API to interact with middleware-specific job management operations.

```
 1  ahe = GObj.create(’org.virolab.ahe’)
 2  call0 = ahe.submit(’sort’, ’sort-job’, 1,
 3                  ’Mavrino’, ’/home/tomek/config.txt’)
 4  call0.wait
 5  # returns a path to a file:
 6  call0.get_result(’/home/tomek/results/’)
 7
 8  sort1 = GObj.create(’org.virolab.Sort’)
 9  call1 = sort1.sort_start(’/home/tomek/input.txt’)
10  call1.wait
11  # returns a path to a file:
12  call1.get_result(’/home/tomek/results/’)
13
14  sort2 = GObj.create(’org.virolab.StringSort’)
15  # returns array [’Ania’, ’Kasia’, ’Tomek’]
16  result = sort2.sort([’Tomek’, ’Ania’, ’Kasia’])
```

Fig. 5. Three approaches to invoking a sample `sort` application accessible with AHE middleware.

To demonstrate the various possibilities of wrapping job-based applications as local gems, we show a simple `sort` application which is a tutorial example of AHE middleware. Fig. 5 presents three approaches to invoking this application. The first one uses the low-level AHE adapter API represented by `ahe` Grid Object (line 1) directly: a user has to specify all the details of the submitted job (lines 2–3), provide files with input and path for storing output and use a generic `submit()` operation. The second Grid Object, `sort1` (line 8), is a local gem which introduces a higher-level `sort_start()` method that accepts a path to an input file and hides some details of AHE usage. The last one, `sort2` (line 14), is a local gem providing a typed method which takes the parameters as array and returns also an array object (line 16): all the details of using AHE middleware are hidden.

As an example of how such a local gem is built, let us consider a `NamdWrapper` class which enables to use the molecular dynamics NAMD [32] application (Fig. 6), which has been installed on a Cyfronet EGEE site. The wrapper uses two classes provided by the Grid Operation Invoker: `GliteResource` and `JobSpec`. The former class uses a middleware specific user interface wrappers (`GliteWMSUIWrapper`) that provide Ruby API to gLite middleware by wrapping the command-line user interface. The latter enables to create job specification and to generate a JDL file for the job.

The wrapper class (Fig. 6) implements the `molecule_simulate` method that hides details about job submission and result retrieving. The method accepts a name of the job, an array with names of input files, an array with names of expected output files and the number of nodes used for running the simulation. In the body of the method a `JobSpec` object is created that includes the information provided as input parameters (lines 5–17). Next, the gLite resource object (line 21) is used to submit the job and obtain a `CallId` object (line 23). This object contains the job id returned by the `glite-wms-job-submit` command and enables to monitor job status and download output and error files (line 26). The path to results of the simulation is returned. If it is beneficial to convert the outcome of job to Ruby objects, this can be done in the wrapper class.

It should be noted, that in this example a specific host (zeus02.cyf-kr.edu.pl) was manually selected by the script developer. In a general case, however, this decision is left to the gLite resource broker; so no additional involvement is required on the part of the script developer. On the other hand, when invoking operations on Grid Objects representing Web Services or components, the GrAppO optimizer module acts as a broker and selects the optimal instance.

```
require 'cyfronet/gridspace/goi/adapters/glite_resource'

class Namd
   def molecule_simulate(jobName, inputs, outputs, nodeNumber)
      jobSpec = JobSpec.new
      jobSpec.executable = '/bin/bash'
      jobSpec.arguments = ('$VO_VOCE_SW_DIR/NAMD_2.6/namd.run alanin.namd')
      jobSpec.stdoutput = 'namd_zeus.out'
      jobSpec.stderr = 'namd_zeus.err'
      outputs.each{|output|
         jobSpec.add_to_output_sandbox(output)
      }
      inputs.each{|input|
         jobSpec.add_to_input_sandbox(input)
      }

      jobSpec.add_property('Requirements','other.GlueCEInfoHostName=="zeus02.cyf-kr.edu.pl"')

      techInfo = {'type' => 'GLITE', 'name' => 'NAMD', 'method#0' => 'submit'}

      namd = GliteResource.new(techInfo)

      callId = namd.submit(jobSpec)
      callId.wait
      if callId.success?
         result = callId.get_result('/home/people/ymbartyn/jobs/outputs')
      elsif callId.failed?
         result = nil
      end
      return result
   end
end
```

Fig. 6. Local class wrapping an EGEE *NAMD* job. It assumes that NAMD package is available on the EGEE infrastructure.

The predecessor of the gLite, LCG/EDG middleware is also supported by the Grid Operation Invoker. `EdgUIWrapper` and `LcgResource` classes provide analogous functionality as those presented in the Namd wrapper example.

7. Asynchronous invocation of Grid Operations

The middleware technologies supported by GOI can offer various interaction modes: either simple, stateless invocations (in the case of Web Services) or a stateful mode (WSRF, MOCCA components) where an instance of a component may be created and then subsequent calls can operate on that particular instance. On the other hand, for job-based middleware, such as gLite or AHE, the natural interaction mode follows the *submit–get_status–get_result* pattern. Additionally, as was shown in the previous section, it is possible to wrap such job-based application within a *local gem* which can provide a single stateless operation, no different from e.g. a Web Service call. In a similar way, it would be worth to add support for *asynchronous* on *non-blocking* invocation mode to the Web Service or component-based operations.

Asynchronous invocation of operations has many advantages: first of all, it allows remote processing to proceed in parallel, which can introduce a significant speedup if multiple resources are available on the Grid. It enables some time-consuming tasks, such as creation of component instances, to run in the background, which can result in performance improvement of experiment execution. Other existing distributed processing frameworks often support asynchronous call of remote operations: NetSolve [22] provides an API for non-blocking calls, the RMIX [33] library offers asynchronous extensions to Java RMI with the usage of *futures* and *callbacks* and by introducing specific naming conventions, whereas ProActive [34] proposes a programming model where all invocations are asynchronous by default and futures are transparent.

```
1   sequence = ">Demo-sequence\nCCTCAAAT ... "
2
3   region = "RT"
4
5   align_tool = GObj.create('regadb.RegaAlignment')
6   subtype_tool = GObj.create('regadb.RegaHivSubtype')
7
8   alignment = align_tool.async_align(sequence, region)
9
10  subtype = subtype_tool.async_subtype(sequence)
11
12  puts "Virus subtype: " + subtype.get_result
13  puts "Alignment result: " + alignment.get_result
```

Fig. 7. Example showing the usage of asynchronous calls to introduce parallel execution of alignment and subtyping tools.

When designing the non-blocking invocation support for Grid Operations in GOI, we decided to introduce an API which could be simple but explicit. Therefore, we proposed a naming convention where adding a `async_` prefix to the method call results in asynchronous invocation. The return of this method is a *future* object which provides `get_result()`, `done?` and `cancel()` methods. An example of the usage of asynchronous invocations is shown in Fig. 7, where two time-consuming tasks can be executed in parallel. In the listing, we can see that `align()` and `subtype()` methods are invoked asynchronously (lines 8 and 10). Synchronization occurs in lines 12 and 13 when the results of operations are requested.

The implementation of asynchronous invocations follows the main design principle of the Grid Operation Invoker, namely to remain unintrusive on the server side. Therefore, all concurrency has to be introduced on the client side and the underlying middleware can remain unchanged. It is important for a script developer to remember that although asynchronous invocations introduce concurrency on the client side, the server-side invocations are unchanged; so the whole state of the call is kept on the client side and the *future* objects are not persistent. For this reason, it is impossible to save them e.g. in a database and check their status in a different execution scenario, which can be done in the case of

```
 1   require 'cyfronet/gridspace/goi/core/g_obj'
 2
 3   namd = GObj.create('cyfronet.gridspace.gem.Namd')
 4
 5   jobName = 'namd'
 6   inputs = ['alanin.namd', 'alanin.params',
 7             'alanin.psf', 'alanin.pdb']
 8   outputs = ['alanin.coor', 'alanin.vel',
 9              'alanin.xsc']
10   nodeNumber = 4
11
12   namd.molecule_simulate(jobName, inputs,
13                          outputs, nodeNumber)
```

Fig. 8. Application submitting a job that executes *NAMD* on the EGEE.

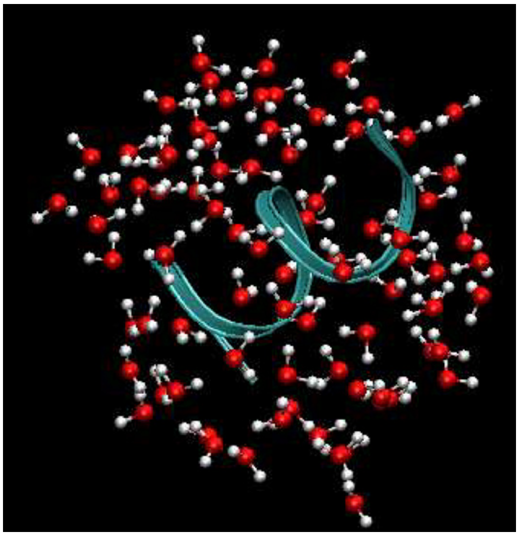

Fig. 9. Visualization of a sample experiment result (alanin molecule in water environment).

job-based middleware, where `call_id` objects represent the job identifier at the server (Grid middleware) side.

8. Application examples

The Grid Operation Invoker was implemented and integrated with the ViroLab Virtual Laboratory. A sample application made accessible there is the NAMD molecular dynamics package. The source code of a sample experiment that performs a molecular dynamics simulation for an alanine amino acid in a water environment is presented in Fig. 8. At the beginning of the script, the developer requires a `GObj` class that provides a uniform interface to create `Grid Object instance` representatives (line 1). Subsequently, in line 3 a representative of the NAMD application is created. Properties of the job are defined in lines 5–10 (job name, input and output files and, finally, the number of nodes used for computation). Line 12 invokes the `molecule_simulate()` method which automatically generates a JDL file for the job, submits the job, monitors its status and retrieves results upon successful completion or informs the user about an error.

This experiment may be further extended by adding invocations of the locally installed VMD (Visual Molecular Dynamics) [35] toolkit (a molecular visualization program) in order to display the obtained results (consult Fig. 9 for a sample view).

GOI is also used to run other experiments in the Virtual laboratory, including a full experiment called "from genotype to drug resistance". It uses gems from RegaDB for HIV genotyping [31] which were integrated using WTS [36] services. Data mining experiments can also be built using the Weka [37] toolkit wrapped as MOCCA components. At present, more services are being added as new gems; among them are the Web Services from the European Bioinformatics Institute [5] and various demonstration services used for education in computational science.

9. Summary and future work

In this paper we have described the Grid Operation Invoker as a tool facilitating application building, used by scientific experiment developers who are familiar with simple script programming. To hide the complexity of details which are usually required to deal with Grid middleware technologies, we have introduced the Grid Object abstraction, which represents any type of computational resource. The support for Web Services and MOCCA components as sample technologies was present from the earliest version of GOI. In this paper we have described how GOI was extended to support Grid Objects representing local processing by *local gems* and how this mechanism can be used to add support for middleware systems based on the batch job submission processing model. By introducing simple wrapper classes implemented in Ruby, we can add an object-oriented interface to various applications executed on EGEE using LCG/gLite middleware as well as those available via AHE. This new technology was tested using the NAMD molecular dynamics package and integrated into the ViroLab Virtual Laboratory.

As the main achievement of our work, we consider 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. Although such multiple levels of abstraction have been earlier applied to workflow systems (e.g. in [11]), there are many scenarios where the scripting approach has clear advantages. Therefore we consider GOI and its application model (with an object-oriented abstraction over Grid middleware) an interesting alternative to graphical workflow-based tools.

Our experience from the development of GOI and the applications that use it allows some general conclusions. First of all, we consider that using Ruby as a high-level workflow language was a good choice since it provides a flexible means of combining co-ordination of remote computing (workflow) with local processing required by experiment scenarios. Additionally, implementing GOI and Grid Objects as Ruby library was convenient due to the dynamic nature of the language. The proposed client-side approach proved useful and practical in the world of heterogeneous middleware, while the Grid Object abstraction was demonstrated to be rich enough to handle diverse technologies and interaction modes. GOI is easily extensible due to its modular architecture and a rich set of libraries that can be used.

The developed system meets both functional and non-functional requirements, although its development involved tackling both conceptual and technical difficulties. The former include merging different interaction models and finding an appropriate set of data for describing instances using diverse middleware packages. The latter concerned resolving conflicts among client-side Java libraries.

We should also mention that although our implementation of GOI is limited to the Ruby language, the main design concepts remain generic. It would be possible to implement GOI in any other dynamic language such as Perl or Python and all other components of virtual laboratory such as Registry or Optimizer could be directly used. GridSpace Engine was successfully adopted in two research projects: GREDIA [38] and the computational chemistry portal *Chempo* in EGEE.

Further work involves analysis, design and implementation of an introspection mechanism that will enable more interactive execution of scripts. This may be useful for collaborative and exploratory programming and will allow the user to immediately see and react to the results of Grid operations performed. We also consider combining the scripting and workflow approaches, so that workflow jobs may be submitted from within a script, or the scripts may be integrated into higher-level workflows. Moreover, we are working on a module enabling us to plug in diverse security mechanisms, such as GSI [39] and Shibboleth [40]. Support for more middleware technologies, such as UNICORE [41] and WSRF [42], is also under development. Thereafter we would like to integrate more *Grid Object instances* to be able to build more complex experiments.

Acknowledgments

This work has been partly supported by the European Commission ViroLab Project [43] Grant 027446, Polish SPUB-M grant, the AGH grant 11.11.120.777, and ACC CYFRONET-AGH grant 500-08. The authors would like to express their gratitude to Tomek Gubala for leading the development of Virtual Laboratory. Thanks also go to Mariusz Sterzel for his hints on integrating NAMD with our system and deployment on EGEE, Stefan Zasada for his help with AHE and to Eryk Ciepiela, Joanna Kocot, Marek Kasztelnik and Piotr Nowakowski for their comments and contributions.

References

[1] G. Aloisio, V. Breton, M. Mirto, A. Murli, T. Solomonides, Special section: Life science grids for biomedicine and bioinformatics, Future Generation Computer Systems 23 (3) (2007) 367–370.

[2] I. Foster, K. Kesselman, Scaling system-level science: Scientific exploration and IT implications, Computer 39 (11) (2006) 31–39.

[3] EGEE Project, Website, 2008. http://public.eu-egee.org/.

[4] DEISA, DEISA Project, 2008. http://deisa.org.

[5] A. Labarga, F. Valentin, M. Andersson, R. Lopez, Web services at the European bioinformatics institute, Nucleic Acids Research 35 (Web-Server-Issue) (2007) 6–11.

[6] ViroLab team at CYFRONET, The ViroLab Virtual Laboratory Website, 2007. http://virolab.cyfronet.pl.

[7] D. Thomas, C. Fowler, A. Hunt, Programming Ruby — The Pragmatic Programmer's Guide, second edition, The Pragmatic Programmers, 2004.

[8] M. Bubak, T. Gubala, M. Malawski, B. Balis, W. Funika, T. Bartynski, E. Ciepiela, D. Harezlak, M. Kasztelnik, J. Kocot, D. Krol, P. Nowakowski, M. Pelczar, J. Wach, M. Assel, A. Tirado-Ramos, Virtual laboratory for development and execution of biomedical collaborative applications, in: Proceedings of the Twenty-First IEEE International Symposium on Computer-Based Medical Systems, IEEE Computer Society, 2008, pp. 373–378.

[9] M. Malawski, T. Gubala, M. Kasztelnik, T. Bartynski, M. Bubak, F. Baude, L. Henrio, High-level scripting approach for building component-based applications on the grid, in: M. Danelutto, P. Fragopoulou, V. Getov (Eds.), Making Grids Work: CoreGRID Workshop on Grid Programming Model Grid and P2P Systems Architecture Grid Systems, Tools and Environments, Springer, Heraklion, Crete, 2008, pp. 307–320.

[10] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, Y. Zhao, Scientific workflow management and the Kepler system, Concurrency and Computation: Practice and Experience 18 (10) (2006) 1039–1065.

[11] F. Neubauer, A. Hoheisel, J. Geiler, Workflow-based grid applications, Future Generation Computer Systems 22 (1–2) (2006) 6–15.

[12] T.M. Oinn, R.M. Greenwood, M. Addis, M.N. Alpdemir, J. Ferris, K. Glover, C.A. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P.W. Lord, M.R. Pocock, M. Senger, R. Stevens, A. Wipat, C. Wroe, Taverna: Lessons in creating a workflow environment for the life sciences, Concurrency and Computation: Practice and Experience 18 (10) (2006) 1067–1100.

[13] T. Bartynski, M. Malawski, T. Gubala, M. Bubak, Universal grid client: Grid operation invoker, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics, 7th International Conference, in: Lecture Notes in Computer Science, vol. 4967, Springer, 2008, pp. 1068–1077.

[14] M. Malawski, M. Bubak, M. Placek, D. Kurzyniec, V. Sunderam, Experiments with distributed component computing across grid boundaries, in: Proceedings of HPC-GECO/COMPFRAME Workshop in Conjunction with HPDC'06, 2006, pp. 109–116.

[15] R. Armstrong, G. Kumfert, L.C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, T. Dahlgren, The CCA component model for high-performance scientific computing, Concurrency and Computation: Practice and Experience 18 (2) (2006) 215–229.

[16] EGEE Project, Lightweight middleware for grid computing, 2008. http://glite.web.cern.ch/glite/.

[17] P. Coveney, R.S. Saksena, S.J. Zasada, M. McKeown, S. Pickles, The application hosting environment: Lightweight middleware for grid-based computational science, Computer Physics Communications 176 (6) (2007) 406–418.

[18] M. Malawski, T. Bartynski, M. Bubak, A tool for building collaborative applications by invocation of grid operations, in: M. Bubak, G.D. van Albada, J. Dongarra, P.M.A. Sloot (Eds.), Computational Science - ICCS 2008, 8th International Conference, in: Lecture Notes in Computer Science, vol. 5103, Springer, 2008, pp. 243–252.

[19] Gridlab Project, Grid application toolkit, 2004. http://www.gridlab.org/WorkPackages/wp-1/.

[20] A.L. Bazinet, D.S. Myers, J. Fuetsch, M.P. Cummings, Grid services base library: A high-level, procedural application programming interface for writing globus-based grid services, Future Generation Computer Systems 23 (3) (2007) 517–522.

[21] K.H. Gjermundrød, M.D. Dikaiakos, M. Stümpert, P. Wolniewicz, H. Kornmayer, g-eclipse — An integrated framework to access and maintain grid resources, in: 9th IEEE/ACM International Conference on Grid Computing, Grid 2008, Tsukuba, Japan, September 29–October 1, 2008, pp. 57–64.

[22] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, S. Vadhiyar, Users' guide to NetSolve V1.4.1, Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[23] C. Zhang, I. Kelley, G. Allen, Grid portal solutions: A comparison of GridPortlets and OGCE: Research articles, Concurrency and Computation: Practice and Experience 19 (12) (2007) 1739–1748.

[24] M. Russell, et al., The vine toolkit: A java framework for developing grid applications, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Wasniewski (Eds.), Parallel Processing and Applied Mathematics, 7th International Conference, in: Lecture Notes in Computer Science, vol. 4967, Springer, 2008, pp. 331–340.

[25] B. Balis, M. Bubak, K. Sterna, A. Bemben, Efficient and reliable execution of legacy codes exposed as services, in: Y. Shi, G.D. van Albada, J. Dongarra, P.M.A. Sloot (Eds.), Computational Science - ICCS 2007, 7th International Conference Beijing, China, May 27–30, 2007, Proceedings, Part I, in: Lecture Notes in Computer Science, vol. 4487, Springer, 2007, pp. 390–397.

[26] T. Delaitre, T. Kiss, A. Goyeneche, G. Terstyánszky, S.C. Winter, P. Kacsuk, Gemlca: Running legacy code applications as grid services, Journal of Grid Computing 3 (1–2) (2005) 75–90.

[27] Z. Zhao, A. Belloum, M. Bubak, Special section on workflow systems and applications in e-science, Future Generation Computer Systems 25 (5) (2009) 525–527.

[28] M. Assel, P. Nowakowski, M. Bubak, Integrating and accessing medical data resources within the ViroLab virtual laboratory, in: M. Bubak, G.D. van Albada, J. Dongarra, P.M.A. Sloot (Eds.), Computational Science - ICCS 2008, 8th International Conference, in: Lecture Notes in Computer Science, vol. 5103, Springer, 2008, pp. 90–99.

[29] B. Balis, M. Bubak, M. Pelczar, J. Wach, Provenance tracking and querying in the virolab virtual laboratory, in: 8th IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2008, pp. 675–680.

[30] P.M.A. Sloot, A. Tirado-Ramos, I. Altintas, M. Bubak, C. Boucher, From molecule to man: Decision support in individualized e-health, Computer 39 (11) (2006) 40–46.

[31] T. de Oliveira, K. Deforche, S. Cassol, M. Salminen, D. Paraskevis, C. Seebregts, J. Snoeck, E.J.J. van Rensburg, A.M.J. Wensing, D.A. van de Vijver, C.A. Boucher, R. Camacho, A.-M. Vandamme, An automated genotyping system for analysis of hiv-1 and other microbial sequences, Bioinformatics 21 (19) (2005) 3797–3800.

[32] J.C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R.D. Skeel, L. Kalé, K. Schulten, Scalable molecular dynamics with NAMD, Journal of Computational Chemistry 26 (16) (2005) 1781–1802.

[33] D. Kurzyniec, V.S. Sunderam, Semantic aspects of asynchronous RMI: The RMIX approach, in: Proc. of 6th International Workshop on Java for Parallel and Distributed Computing, in conjunction with IPDPS 2004, IEEE Computer Society, Santa Fe, New Mexico, USA, 2004.

[34] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, R. Quilici, Programming, deploying, composing, for the grid, in: J.C. Cunha, O.F. Rana (Eds.), Grid Computing: Software Environments and Tools, Springer, 2006.

[35] W. Humphrey, A. Dalke, K. Schulten, VMD — Visual Molecular Dynamics, Journal of Molecular Graphics 14 (1996) 33–38.

[36] P. Libin, Witty Services, 2007. http://wts.sf.net.

[37] I.H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, Morgan Kaufmann, San Francisco, 2005.

[38] P. Nowakowski, D. Harezlak, M. Bubak, A new approach to development and execution of interactive applications on the grid, in: 8th IEEE International Symposium on Cluster Computing and the Grid, IEEE Computer Society, 2008, pp. 681–686.

[39] I.T. Foster, C. Kesselman, G. Tsudik, S. Tuecke, A security architecture for computational grids, in: ACM Conference on Computer and Communications Security, 1998, pp. 83–92.
[40] Internet 2 Project, Shibboleth, 2008. http://shibboleth.internet2.edu/.
[41] Unicore Forum, Unicore, 2008. http://www.unicore.org.
[42] Globus Alliance, The WS-resource framework, 2007. http://www.globus.org/toolkit/.
[43] ViroLab Project Consortium, ViroLab, 2008. http://virolab.org.

Maciej Malawski has a Ph.D. in Computer Science (2009, AGH, Krakow, Poland), M.Sc. in Computer Science (2001, AGH, Krakow) and in Physics (2004, Jagiellonian University, Krakow). Since 2001 he has been a research and teaching assistant at the Institute of Computer Science AGH. He was involved in EU-funded projects such as CrossGrid, K-WfGrid and CoreGRID at ACC CYFRONET AGH. During 2004–2005 he was a research fellow at Distributed Computing Laboratory at Emory University, Atlanta. Currently he is involved in the EU IST ViroLab project, where he is responsible for component-based middleware. His scientific interests include parallel computing, Grid systems, distributed service- and component-based systems, and e-science applications.

Tomasz Bartynski is a computer scientist at the Academic Computer Center CYFRONET AGH in Krakow. He received his M.Sc. degree in computer science from the AGH University of Science and Technology in Krakow. Since 2006 he has been working for ACC CYFRONET AGH in Krakow. Currently he is involved in EU-funded projects: ViroLab and GREDIA as a designer and developer. His scientific interests involve distributed systems, parallel computing, Web technologies and programming paradigms.

Marian Bubak has an M.Sc. degree in Technical Physics and Ph.D. in Computer Science. He is an adjunct at the Institute of Computer Science and ACC CYFRONET - AGH University of Science and Technology, Krakow, Poland, and a Professor of Distributed System Engineering at the Informatics Institute Universiteit van Amsterdam. His research interests include parallel and distributed computing, Grid systems, and e-science; he is the author of about 200 papers in this area, co-editor of 15 proceedings of international conferences and the Associate Editor of FGCS Grid Computing. In the CrossGrid Project he was the leader of the Architecture Team, the Scientific Coordinator of the K-WfGrid, the member of the Integration Monitoring Committee of the CoreGRID, and the WP leader in ViroLab and GREDIA EU IST projects.