Akademia Górniczo – Hutnicza
im. Stanisława Staszica
w Krakowie
Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki

Katedra Informatyki

Tomasz Bartyński

Zdalne wykonywanie zleconych operacji z użyciem różnorakich protokołów komunikacyjnych dobieranych automatycznie

Praca magisterska

Kierunek: Informatyka
Specjalność: Systemy rozproszone i sieci komputerowe

Promotor:
dr inż. Marian Bubak

Konsultacja:
mgr inż. Maciej Malawski
mgr inż. Tomasz Gubała

Nr albumu: 116894

Kraków 2008

Oświadczenie autora

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Tomasz Bartyński

AGH University of Science and Technology
in Kraków

Faculty of Electrical Engineering, Automatics, Computer Science
and Electronics

Institute of Computer Science

Tomasz Bartyński

Remote execution of delegated operations with support for automatic selection among multiple communication protocols

Thesis

Major: Computer Science
Specialization: Distributed Systems and Computer Networks

Supervisor:
dr. Marian Bubak

Consultancy:
Maciej Malawski
Tomasz Gubała

Album id: 116894

Kraków 2008

Oświadczenie autora

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i że nie korzystałem ze źródeł innych niż wymienione w pracy.

Tomasz Bartyński

Abstract

This thesis presents a novel approach to development of grid applications which address the issue of solving highly complicated scientific problems which may require great amount of computational power or storage as well as gathering and combining results of various programs executed in distributed environment. An experiment may contain application logic, which can not be easily expressed in currently available tools.

This work is focused on finding an appropriate model of grid programming that would allow fast and easy development of high level application, which are able to take advantage of grids, with no limits in expression of experiment logic. Our solution is based on a client-side interface to grid environment which can be accessed within modern scripting language. This approach facilitates reusing existing software, which is already published, as well as harnessing the computation power of a grid environment.

This thesis is organized as follows: First, we provide the background on the grid environment and obstacles associated with using it, which is the motivation for this work. Next, similar solutions are analyzed. Then our approach, which is based on accessing the grid from within a scripting language, is introduced. It is followed by the design and description of the implementation of the Grid Operation Invoker system. Finally we present the results of appling our tool to develop high-level grid applications.

Key words

high level grid programming, grid middleware, uniform interface to the grid environment, remote operation invocation, JRuby, object-oriented scripting language

Acknowledgments

First of all, I would like to express my gratitude to my supervisor, dr. Marian Bubak, for guidance, patience and invaluable advices. I would like to sincerely thank Maciej Malawski and Tomasz Gubała for their support in design of the system and implementation counsels.The author also wishes to acknowledge contributions from his colleagues from ACC Cyfronet AGH, including Joanna Kocot, Eryk Ciepiela, Marek Kasztelnik, Piotr Nowakowski and Daniel Haręźlak. Finally I would like to thank my professors and all my colleagues I met during my education at AGH university.

Without those people this work would not be what it is.

This work has been made possible through the support of the European Commission ViroLab Project [1] Grant 027446.

www.agh.edu.pl www.cyfronet.pl www.virolab.org www.virolab.cyfronet.pl

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

Abbreviation or acronym	Explanation	First occurrence in section
WS	Web Service	1.1
MOCCA	a CCA compliant distributed component framework	1.3.1
WSRF	Web Services Resource Framework	1.3.1
RMIX	a Java communication framework based on the RMI paradigm	1.3.1
RMI	Remote Method Invocation	1.3.1
UI	User Interface - a set of programs enabling job submission on a grid infrastructure, like EGEE	1.3.1
LCG	LHC Computing Grid	1.3.1
gLite	Grid middleware created within the EGEE project	1.3.1
Unicore	Uniform Interface to Computing Resources - client and server software for grid computing	1.3.1
SOAP	Simple Object Access Protocol	1.3.1
RPC	Remote Procedure Calls	1.3.1
WSDL	Web Service Description Language	1.3.1
JDL	Job Description Language (used by LCG)	1.3.1
AJO	Abstract Job Object (used by the Unicore middleware)	1.3.1
API	Application Programming Interface	1.3.1
W3C	The Worl Wide Web Consortium	2.2.1
XML	eXtensible Markup Language	2.2.1
EPR	Endpoint reference	2.2.1
WTS	Witty Services	2.2.3
MoML	Modeling Markup Language	3.2.1

WSIF	Web Service Invocation Framework	3.1
JSDL	Job Submission Description Language	2.2.3
GEODISE	Grid Enabled Optimization and Design Search for Engineering	3.1
K-Wf	Knowledge-based Workflow System for Grid Applications	3.1
GAT/SAGA	Grid Application Toolkit/Simple API for Grid Applications	3.1
EJB	Enterprise JavaBeans	3.3.4
JMS	Java Message Service	3.3.4
DRMAA	Distributed Resources Management Application API	3.4
GWorkflowDL	Grid Workflow Description Language (used in the K-Wf system)	3.4
GOI	Grid Operation Invoker	4.2
UML	Unified Modeling Language	5.2
HIV	Human Immunodeficiency Virus	8.2

Table 1. Abbreviations and acronyms

Chapter 1

Introduction

This chapter presents the motivation and objectives of this thesis and defines the problem that we try to solve. It starts with a short description of a grid environment which is followed with introduction of the concept of accessing remote resources from a scripting language. Next, difficulties in using and accessing grids are explained. Finally, we list the work that is required to solve the problem.

1.1. The Grid Environment

Nowadays, researchers from plethora of domains of exact and natural science investigate highly complicated problems. Some of them already take advantage of the in-silico experiments. Such studies gained the approval of the scientific community, constitute a significant part of modern research and will become even more attractive for scientist [2]. In-silico experiments may require large amounts of computational power or storage. Additionally, they often involve complex and specialized software tools. It is essential to reuse the existing software, because creating new software from a scratch is highly expensive and time consuming process. Moreover, utilizing proven libraries and tools increase reliability and efficacy of an application. Finally, collaboration between experts from a diversity of domains can be crucial for a successful work. All these factors moved the computation from local machines to a distributed environment and produced an abundance of challenges for computer scientists.

Grid technologies originated to satisfy these requirements [3]. They allow resource aggregation and virtualization, in order to deliver greater computational power and storage to endusers. Besides that, grid middleware technologies facilitate publishing and reusing software. A wide range of middleware technologies were developed, among which the most accepted are job oriented middlewares, Web Service (WS) [4], stateful services (WSRF) [5] or component-based middlewares, each enabling different interaction and programming models. Needless to say, any of them is suitable for all users, due to their vast variety of requirements. Usually, middleware technologies employ different communication protocols, what prevents interoperability among them. Furthermore, some of them request credentials in a specific format. It gets even more complicated, because the grid environment is heterogeneous and dynamic. Resources are distributed all over the world in a variety of independent administrative domains. Resource pool may change and the load of each machine is fluctuating.

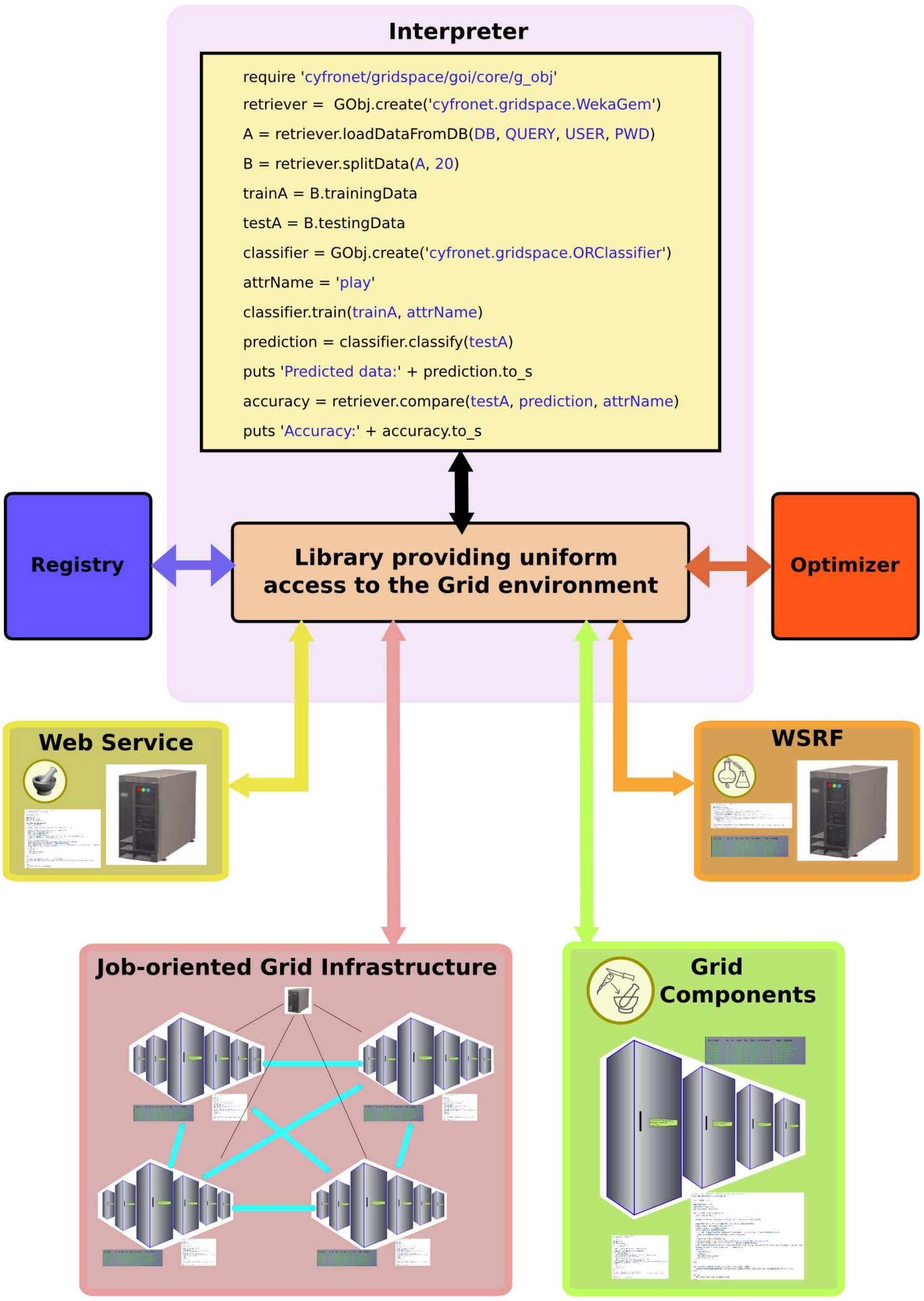
As a consequence, grid usage is difficult from enduser's point of view, as well as for a developer implementing experiments. Currently, there are lots of efforts in providing more user friendly access to grids, for instance through a portal. Besides that, multitude of work is ongoing to produce efficient tools facilitating experiment development. Most of such projets are based on workflow engines, which we believe is not a good solution, if the experiment contains a more elaborated application logic or algorithm. In our opinion there is a necessity for a mechanism that will allow fast and easy development of high level applications accessing grid resources.

1.2. Accessing grids from a scripting language

In our opinion, it is crucial that developing and running high-level grid applications is as easy as creating software that is run locally. Application developers should be focused only on the problem and the solution. It is important to enable them to use best practices and patterns in the application development, for instance the object-oriented programming paradigm. They should be able to use objects representing remote software in the same manner as ordinary objects, instead of being concerned about the obstacles associated with invocations of remote operations. Ideally, developer should only request for functionality, by selecting an appropriate class, rather than interfacing directly remote software in the source code. The process of finding remote realization of desired functionality, which fits best user's needs, ought to be automated and transparent for the developer.

Modern scripting languages are an interesting alternative for compiled languages. Due to interpretation line-by-line and being untyped, they are especially suitable for fast prototyping and developing high-level applications adapting to the dynamic grid environment at run-time. Languages, such as Ruby [6], Perl [7], Python [8], are broadly accepted in the world of computer science and proved their usability in the area of developing universal applications. These languages have dedicated reliable interpreters for all platforms, good support for them is provided both on the Web and in numerous publications. Besides that, they support object-oriented paradigm and thus allow creating complicated, yet well structured and clear projects.

We are strongly convinced that the solution addressed at the defined problem should be based on a modern, object-oriented, scripting language, which ought to be extended with the capability to access the grid environment. It is required from it to allow use of the computational resources in a coherent and transparent manner from the developer's point of view. Such approach combines the advantages of enabling fast development of high-level application and harnessing the potential of grids. Fig. 1.1 illustrates the idea of a script being a high-level grid application.



```
require 'cyfronet/gridspace/goi/core/g_obj'
retriever =  GObj.create('cyfronet.gridspace.WekaGem')
A = retriever.loadDataFromDB(DB, QUERY, USER, PWD)
B = retriever.splitData(A, 20)
trainA = B.trainingData
testA = B.testingData
classifier = GObj.create('cyfronet.gridspace.ORClassifier')
attrName = 'play'
classifier.train(trainA, attrName)
prediction = classifier.classify(testA)
puts 'Predicted data:' + prediction.to_s
accuracy = retriever.compare(testA, prediction, attrName)
puts 'Accuracy:' + accuracy.to_s
```

Interpreter
Registry
Library providing uniform access to the Grid environment
Optimizer
Web Service
WSRF
Job-oriented Grid Infrastructure
Grid Components

Figure 1.1. Overview of the concept of a script being a high-level grid application utilizing various middleware technologies.

1.3. Description of the problem

1.3.1. Difficulties in accessing grid resources

Capabilities of geographically dispersed, heterogeneous computational and storage resources are delivered to endusers using, in most cases, one of the wide range of middleware suites. A set of services is installed on the resources in order to make them accessible for each other and for endusers. Installed middleware implies the way how resources are being accessed. Each technology provides access to computation in its specific manner. Web Service, WSRF and component-based middleware suites, like MOCCA [9], provide access through client-side software, which needs to be developed by the user, while other provide client programs to submit jobs, User Interface (UI) for LCG [10] and gLite [11] or a Java based Unicore [12] client. In the latter case users need to execute few commands to submit a job and retrieve results after completion of the job. Their attention is required to watch job status during execution. In the former case lots of developers' effort is spent only on accessing the resource instead of solving the problem. Further more, grid users need to gather information about service availability and load in order to select the resource that best fits their requirements.

Interoperability among middleware technologies is prevented due to numerous reasons. First of all, different communication protocols are used, for example Web Service and WSRF use SOAP [13], while MOCCA employs RMIX [14], which is an extension of the RMI protocol. Next, different security mechanisms and credentials are used. Usually, every middleware suite requires a specific set of libraries or tools, which can be large in size and non trivial to install. Finally, inputs and results are in various forms and formats (files, objects, XML [15] documents), therefore passing results as arguments between middlewares without transforming them is not possible.

Besides described obstacles many middleware suites are under active development and theirs Application Programmer Interfaces (APIs) are not stable. Some of existing suites did not, and will not, achieve production status.

Nowadays creating and running applications accessing grids are time consuming and hard. It requires good understanding of underlying middleware suite because lots of low-level programming or usage of dedicated tools are required. What is more, any of existing upper level system (please refer to section 3.1 for a brief discussion on existing solutions) provides convenient access to many technologies, nor allows interoperability. These facts motivate us to conduct research in the scope of unifying the interface for accessing grid resources, and to develop a prototype as well.

1.3.2. External information dependency

Every automated system capable of accessing the dynamic grid environment requires both, static and dynamic information about it. The static data includes:

- a list of resources and theirs unique identifications,
- endpoint addresses (of services, components or UI machines),
- operation signatures (inputs, outputs, data formats etc.),
- a list of operations that is provided by each computing resource,
- technology/protocol specific data (for instance: component based middleware suites require the following data: class of the component, component port, codebase etc.; Web Services the type of service (RPC or document), location of the wsdl; jobs need to be described in JDL or AJO).

The dynamic information provides data about:

- failures,
- resource availability,
- load of the machines.

1.3.3. Dependency on external components

Information about the resources needs to be stored and delivered on demand. Record of static data can be kept by a simple class with hard-coded information if the amount of data is small in size and it will be used locally. In other case, registry should be a standalone and independent system providing remote access to static data. Dynamic data about computing resource availability or load needs to be collected periodically and updated in the registry. Such data allows selecting the resource that meets best users needs in terms of computation speed, accuracy or other requirements. Such selection should be made automatically without user attention. Local optimizer can use one of the simplest algorithms, such as random or round-robin, however if quality of the selection is important, more complicated algorithms should be employed and an external optimizer should be used.

1.4. Objectives of the thesis

The main objective of this thesis is to develop a computer system providing uniform access to application elements which are distributed as components, web services, etc. on clusters and grid systems within a scripting language. The system will allow taking advantages of programming features offered by various middleware suites and emulate some of them. This will involve:

- state-of-the-art in the existing middleware technologies, programming models and high-level solutions in order to choose the most useful middleware suites and programming models, as well as to reuse valuable ideas and software,
- in-depth analysis of functional and non-functional requirements for the system,
- detailed design of the system (architecture, data flow and implementation technology),
- development of prototype and adding successively support for more technologies,

- validation of the system, performance testing and discovering the bottlenecks of the system and adding optimization if required,
- listing possibilities of enhancing the system in the future,
- providing documentation on how to install and use the system.

This thesis is structured with regard to these objectives and consists of the following chapters:

1. **Introduction** that provides background on the grid environment and discusses the concept of accessing grids from a scripting language,
2. **Overview of programming models and middleware technologies** that povides a bief overview of leading middleware technologies and their programming properties,
3. **Systems for building grid applications** which analizes high-level systems providing access to grids and discusses their advantages and drawbacks,
4. **Vision of a Grid Operation Invoker** that introduces abstraction over grids and illustrates the scripting approach to building grid applications,
5. **Analysis phase of the Grid Operation Invoker** that collects all use cases and requirements of a system realizing concepts from the previous chapter,
6. **Design and implementation of the Grid Operation Invoker** that provides description of the architecture, principle of operation and implementation of the GOI system,
7. **Validation of the Grid Operation Invoker** which confronts implemented system with requirements,
8. **Grid applications using the Grid Operation Invoker** that presents real-life applications utilizing the GOI library,
9. **Summary and Future Work** that summarizes the thesis and describes possible evolution of the GOI system.

The thesis inludes also the following appendices **Installation guide**, **Grid Operation Invoker API**, **Implementing technology adapters**, **Technology information stored in Registry** and **Publications**.

Chapter 2

Overview of programming models and middleware technologies

This chapter gives a brief overview of programming properties and middleware suites. Programming models and interaction modes are discussed in context of various middleware technologies.

2.1. Basic properties of programming models

Many programming models, paradigms and concepts have evolved during past years in the area of distributed computing. Various middleware suites provide distinct programming features. First of all, middleware technology can support object-oriented paradigm. In such a case, developer implements an object-oriented application and create an object that represents a remote software entity and provides its functionality as methods. Besides that, developer of the high-level grid application is enabled to interact with remote software in either stateless manner, in which case the state is not preserved on the server-side, or in stateful manner. Moreover, operations can be invoked in a synchronous, blocking way, or in an asynchronous manner. The latter allows concurrent execution of more than one operation, thus augments the efficiency of a program, which can be run on many distributed resources simultaneously. Another distinction can be made based on the criteria whether the software being published is a fully functional, independent and self-sufficient entity or is a set of units with defined interfaces and dependencies that can be composed into various applications. The last feature, that is taken into account in this work, is the share mode, which enables to decide if a software entity can be either public or private. Public one is visible for other users and can be used by them, while a private one is visible and accessible only by its owner.

We are strongly convinced that experienced developers should be provided with a uniform API to access the resources, however programming properties should be exposed to them. Skillful use of them can result in a high-level applications of better quality in terms of efficacy and security.

2.2. Middleware

Grid middleware is a software that provides abstraction over heterogeneous and distributed resources. It allows interoperability among diverse platforms as well as among machines coming from different vendors. Middleware packages usually consist of a set of enabling services and can be viewed as a virtualization layer between operating system of each resource and application. Middleware facilitates building distributed applications by providing mechanism for multiple processes to interact with one another across the network.

Middleware packages vary in type. Job-oriented infrastructures are complex systems among whose resources coordinating nodes and working nodes are distinguishable. The former have a set of services installed that are responsible for managing the resource pool, which includes gathering and publishing some dynamic information, scheduling and brokering. The actual computation is performed on one or more machines from the worker node pool. Modern technologies, such as Web Services, WSRF or component-based technologies, use containers installed on every machine that constitutes a distributed environment for applications. Every resource that has a container provides computational power to endusers. There is no need for a central manager (or broker), which is necessary in a job-oriented middlewares.

2.2.1. Service-oriented middleware

Web Service

Following the definition by the W3C [4]:"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

This technology gained acceptance among industry and academic communities. SOAP [13] proving a RPC semantics, combined with XML [15] allow language and platform independence. WSDL [16] provides a standardized mechanism for describing interfaces of services, thus facilitates service discovery and matchmaking. Web Service technology allows stateless, synchronous interaction. Numerous frameworks and libraries have been created to support publishing software as a service in programming languages, such as Java, C, Perl, Python or Ruby.

All mentioned features makes this technology highly-usable for development of high-level grid applications, therefore the WS middleware is in scope of our interest.

Web Services Resource Framework

WSRF [5] is a set of specifications defining a generic and open framework for modeling and accessing stateful resources using Web services. State of such a resource, called *WS-Resource*, is modeled by a XML based *Resource Properties document*, which is referenced in the WSDL. *WS-Addressing* endpoint reference (EPR) contains both address of the service and identifier of the resource. All specifications of the WSRF can be found on the OASIS web page devoted to this standard [5].

This technology is not as accepted as Web Service, though there are many implementations of this standard. The most popular are the Globus Toolkit 4 [17], providing Java and C WS-cores, and Apache WSRF [18] providing a Java implementation. Besides these, there are WSRF::Lite [19] for Perl, pyGridWare [20] for Python and WSRF.NET [21] for .NET.

WSRF is significant from our point of view, because it introduces the idea of stateful interaction with Web services.

2.2.2. Component-based middleware

Alternative for service-oriented middleware suites are component-based technologies. The foundation of such an approach is "composition of applications from software units with specified interfaces and dependencies. The components can be deployed independently and can be composed by a third party" [22].

GridCCM

The GridCCM [23] is an attempt to adapt CORBA Component Model (CCM) for scientific applications by providing efficient implementation of CORBA and parallel extensions, for instance, support for MxN component interactions. Deployment of component applications on grids, including planning and execution phases, is done by means of the ADAGE tool.

ProActive/Fractal

The ProActive [24] introduces approach to building component frameworks based on active objects. It is a Java distributed component framework for parallel application that can be executed within multi-core processors, distributed on Local Area Network (LAN), on clusters and data centers, on intranet and Internet grids. It is based on the Fractal [25] component model, which allows hierarchical component composition.

H2O plus MOCCA

MOCCA [9, 26] combined with H2O is in our opinion a very interesting component-based middleware suite. MOCCA is a component framework compliant with the CCA [27], which adapt the component model to high-performance scientific computations. MOCCA Light, which a Java implementation of MOCCA, is built on top of the H2O [28, 29] platform, which provides lightweight containers called kernels. What is distinguishable in this solution is the separation of the resource provider and service provider roles, because authorized users are allowed to deploy and run their code as pluglets within containers running on resources provided by third parties. Each pluglet is being executed in a separate environment, thus security is ensured, and communicate with other pluglets via RMIX protocol, which extends standard RMI protocol.

We consider such an approach as attractive from the point of view of grid user. The ability to deploy components at run-time on third party resources only by providing URL of the code base seems to be a very significant feature from the high-level application developer perspective. What is more, components enable to interact in a stateless and a stateful manner, as well as allow creating private entities.

2.2.3. Job-oriented middleware

Although the trend in grid computing is towards a service oriented architecture or component-based middleware suites, job-oriented software is still very common in high-throughput computing. Batch processing systems are the most proven and reliable technologies that are deployed on the world's largest infrastructures, such as EGEE [30] or DEISA [31].

LCG

Large Hadron Collider Computing Grid (LCG) [10] is a system for building grids. It is based on a Globus Toolkit 2.4 and a Condor-G. It consists of a set of components that have well defined functionality:

- Workload Management System (WMS) is responsible for matching jobs requirements to the available resources, scheduling the job on an appropriate computing element, checking the job status and retrieving output files;
- Data Management System (DMS) provides file management functionality;
- Information System (IS) gathers and publishes various information on resources;
- Authorization and Authentication System;
- Accounting System;
- Various monitoring and installation services.

LCG provides access to the world's largest grid production status infrastructure-EGEE, therefore it is well tested and documented. On the client-side, command-line software, EDG User Interface, is used, which provides commands for job submission and management and as well as for data manipulations. This solution, however is difficult to install on a client machine. Usually there are dedicated machines with the UI software installed and user logs to this machine to submit jobs. Moreover, LCG does not follow modern trends in distributed computing.

gLite

gLite [11] is superseding LCG at EGEE infrastructure. It is based on the LCG 2.7 but it is a more lightweight middleware package and it is more Web Service-centered. gLite has two layers:

- High-Level Grid Services, which is is not mandatory, allows users to build computing infrastructures;

- Foundation Grid Middleware mupagest be installed on the infrastructure (EGEE) to provide complete and robust middleware.

Being deployed on EGEE infrastructure and reusing reliable components of LCG middleware are advantages of the gLite. It is guarantied that it will be supported in future and its quality will improve. On the other hand, gLite is a hybrid. It is evolving towards service-based technologies, but it depends on the LCG components. gLite also requires an user interface to be installed on a client machine, therefore it is not as light-weight as it was supposed to be.

UNICORE

UNICORE [12] stands for Uniform Interface to Computing Resources. It provides access to distributed, heterogeneous resources in a secure, convenient and uniform way via a user-friendly graphical interface. This middleware consists of:

- End-user interface which is a Java program enabling user to create and monitor jobs, handling complex workflows and managing files and certificates. It has a a user-friendly graphical user interface.
- Server Tier including gateways, which are single points of entry to a Unicore site, and Network Job Supervisors(NJS), which virtualizes heterogeneous resources, such as a single supercomputer or Linux cluster, by representing them as virtual sites. NJS maps the abstract job onto a target system (incarnation) using system-specific data, stored in the Incarnation Data Base (IDB).
- Target System Interface (TSI), which is a stateless daemon running on the target system that provides implementation of the interface to the underlying resource.

Client program communicates with gateway using SSL sockets. Jobs are represented using Abstract Job Object (AJO), which is a system-independent job description. UNICORE is a mature technology, which is deployed on DEISA. Its main drawback is the fact that there isn't any interface providing RPC semantics for executing operations on grids.

GridSAM

GridSAM [32] is an open-source project that introduce the concept of providing job submission and monitoring functionality through a Web Service. It has a modular architecture, therefore the system can interface a wide range of Distributed Resource Managers (DRM) and can be extended with plugins for job submission and file transfer by third parties. Standardized language JSDL, defined by the Global Grid Forum, is used to describe jobs. The Web Service API of GridSAM can be embedded into grid applications, thus usage of distributed resources in an object-oriented manner is enabled, which is a step forward in development of high-level grid applications. GridSAM however, provides access only to job-oriented technologies.

Witty Services

Witty Services (WTS) [33] enables to manage jobs through a Web Service. It is based on an extension of the popular Web Service framework - Apache Axis - build in accordance with the KISS(Keep It Simple, Stupid) principle. It enriches it with the capability of keeping the state of the resource. WTS contains both the server-side Java package and the client side software available for Java and C# developers. The unquestionable advantage of this solution is its simplicity compared to WSRF frameworks.

2.3. Programming properties provided by each middleware technology

Having analyzed most accepted middleware technologies, their features and advantages, we distinguished qualities facilitating development of high-level grid applications. Programming properties, that we have found the most significant are listed below:

- support for object-oriented programming paradigm,
- composability into larger applications,
- stateless and/or stateful interaction mode,
- ability to execute operations concurrently,
- synchronous and/or asynchronous invocation of operations,
- ability to choose public or private sharing mode.

These properties and selected middleware suites providing them are presented as a technology/property matrix in table 2.3. One can observe that properties which need to be added by our system are the asynchronous invocation of operations for service-oriented and component-based technologies and support for object-oriented paradigm for batch processing suites.

Technology	**Programming property**							
	Object-oriented	Composable	Interaction mode		Concurency	Operation invocation		Share mode
			Stateless	Stateful		Synchronous	Asynchronous	
Web Service	yes	no	yes	no	yes	yes	added	public
WSRF	yes	no	yes	yes	yes	yes	added	public
WTS	yes	no	yes	yes	yes	added	added	public
MOCCA	yes	yes	yes	yes	yes	yes	added	private or public
LCG	added	no	yes	no	yes	no	yes	public
gLite	no	no	yes	no	yes	no	yes	public
UNICORE	no	no	yes	no	yes	no	yes	public
GridSAM	yes	no	yes	yes	yes	yes	no	public

Table 2.1. Middleware technology/programming properties matrix. Properties that are offered by a middleware suite itself are marked *yes* in the table, the ones that are emulated by our system on the client side are indicated *added*.

Chapter 3

Systems for building grid applications

In this chapter we discuss high-level systems that enable to build and execute applications, which can be expressed either as workflows or in programming languages, that access the grid environment. We focus on those aspects that we found relevant for our system. We discuss advantages and drawbacks of each solution and finally, we present conclusions.

3.1. High-level systems for accessing grid resources

In this chapter we focus on solutions that allows building grid applications using abstraction over middleware technologies mentioned in Chapter 2 and executing them. High-level systems should hide the complexity of the grid environment and internals of invoking operations on distributed computational resources. These systems providing access to the grid environment can be classified in two groups, based on an approach taken to expressing an application. These groups are:

1. workflow-based systems;
2. libraries, frameworks or other extensions enabling to use grid resources within programming languages.

The former group includes Kepler, Triana an K-Wf Grid while GAT/SAGA, GEODISE, NetSolve/GridSolve and WSIF are members of the latter.

3.2. Workflow-based systems

Workflow is a sequence of steps that are executed in order to achieve some processing intents. Every step represents one operation that does some processing on its inputs to produce an output. Developers create applications using graphical tools that enables to build workflows of blocks, which represent processing, and connections between these blocks, which repersents flow of data and control.

3.2.1. Kepler

This open-source project provides [34] a platform for constructing and executing scientific workflows. It enables to use the "drag and drop" method to build workflows, which are expressed in a MoML language (Modeling Markup Language). Kepler features actor-oriented approach. An actor is a single step in a wokflow. It represent one operation that is performed on its inputs to produce output. Actors can access Web and Grid Services, Globus Grid Jobs or use GridFTP operations. They are connected with one another by ports, that either produce or consume some data. Additional relations can be defined to direct the output port to many actors. Workflow execution is controlled by a Director, which enables a particular model of computation to be used. Kepler platform supports nested workflows (a workflow can be an actor).

This project introduces interesting concepts, for instance nested workflows, ease of expressing experiments and support for more than one middleware technology. In addition, this platform can be extended by adding new actors.

3.2.2. Triana

Triana [35] is a problem solving environment with a similar approach as Kepler. It provides an intuitive user-friendly tool for constructing workflows and an execution engine. It allows combining local operations, Web Services and grid jobs in a single workflow, as well as supports dynamic Web Services discovery and invocation. Besides that it can submit jobs (Globus, Gridlab) using GAT and supports P2PS services. This solution has limitations of every workflow-based system (see Section 3.4).

3.2.3. K-Wf Grid

Knowledge-based Workflow System for Grid Applications [36] (K-Wf) enables to construct workflows in an abstract manner and execute them in the grid environment. It facilitates composing workflows by means of ontology-based semantic reasoning. Besides that, it enables users to monitor the performance, analyze the resulting monitoring information and finally to reuse the joined knowledge of all participants in a collaborative way in order to efficiently construct workflows for new grid applications.

The most interesting feature of this system, from our point of view, is the multiple level of abstraction of describing the workflow. Users can express theirs request in a formal manner. These descriptions are used to automatically build abstract workflows. There may exist a wide range of service candidates capable to perform the requested computations. Among them, the optimal services are selected and resources allocated. This process is depicted in Fig. 3.1 [1].

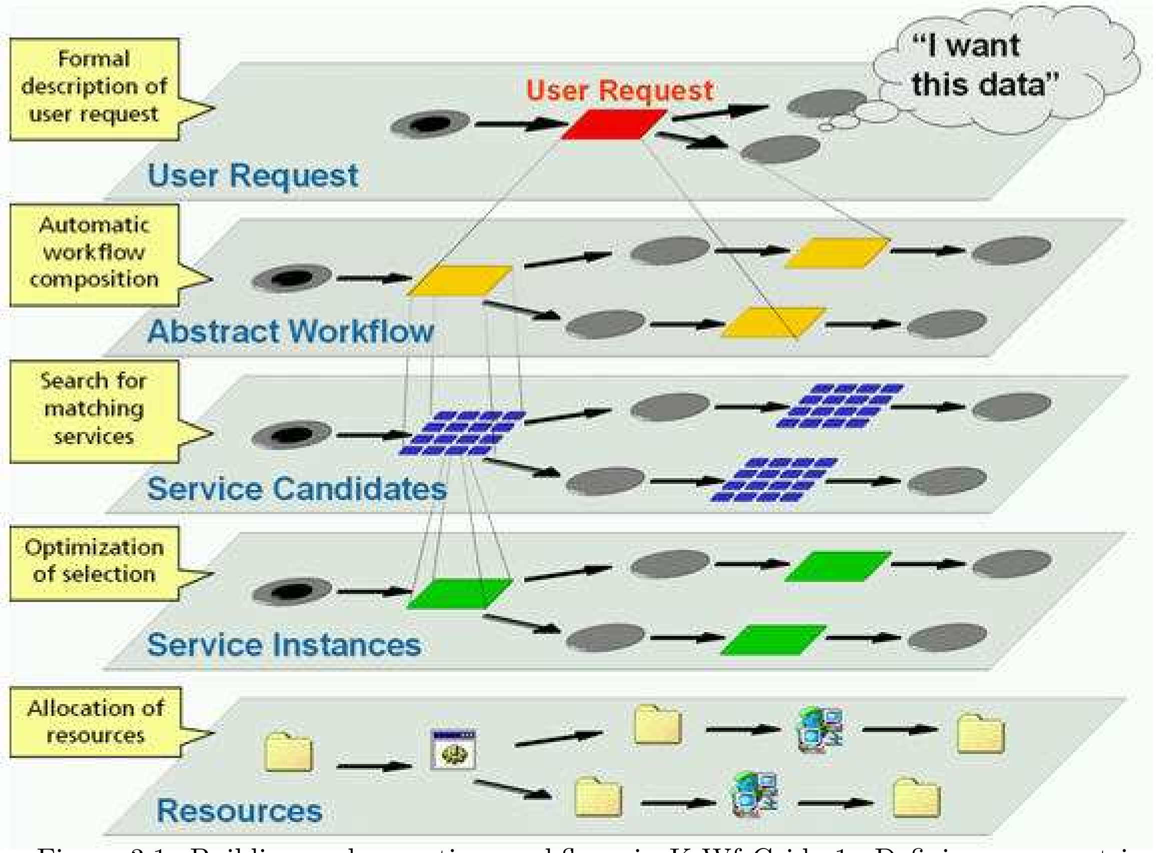


Figure 3.1. Building and executing workflows in K-Wf Grid: 1. Defining a request in a formal manner. 2. Composing an abstract workflow. 3. Search for matching services. 4. Optimization of selection. 5. Allocation of resources.

3.3. Libraries and framework enabling to access grid resources

There are many libraries and frameworks trying to facilitate access to heterogeneous grid resources. They introduce various approaches and provide different features. This section briefly discusses the most interesting ones.

3.3.1. GAT/SAGA

Grid Application Toolkit [37] is an attempt to solve the "many Grids, little applications" problem. It provides a simple,invariant and language-neutral API for accessing grid resources from high-level grid applications, portals or other systems. Operations included in the API include basic use cases, such as file manipulations,

[1] Image from *http://www.kwfgrid.eu*

monitoring and events, managing resources and jobs, information exchange. Moreover, GAT allows error handling and ensures security. GAT objectives are simplifying grid applications, enabling code reusability, making the code more concise and facilitating software maintenance. Further more, GAT makes applications less vulnerable to changes in middleware suites. This toolkit consists of the API, adaptors dedicated for specific infrastructure that implement the API, and an engine that selects an appropriate adaptor at runtime and provides error tracing and fallback mechanisms. Fig. 3.2 [2] present GAT as a part of high-level system (GridLab).

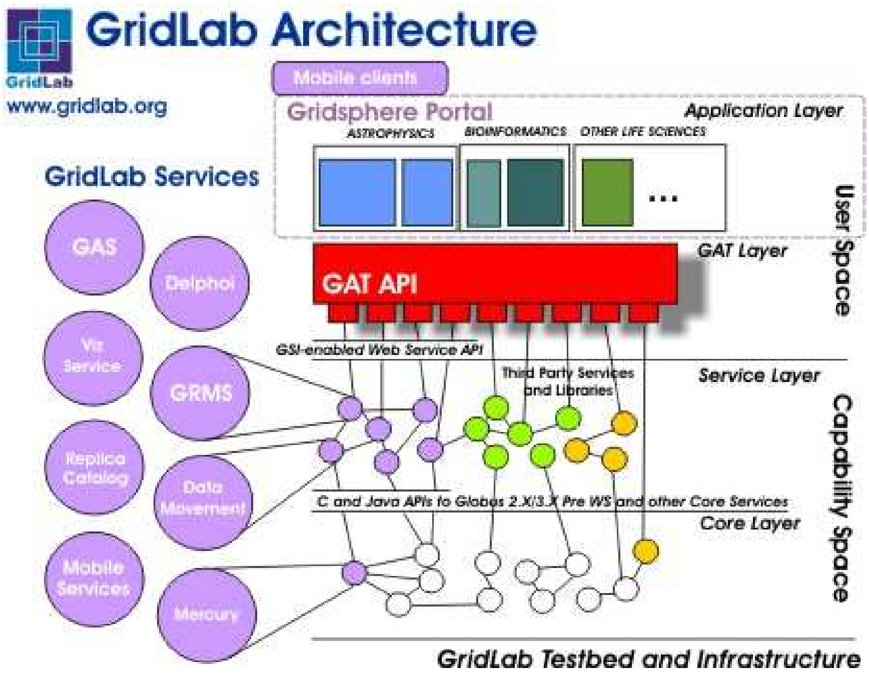


Figure 3.2. Grid Application Toolkit inside the GridLab. GAT provides an API for applications and portal as well as its implementations to underlying grid technologies as adaptors.

GAT library is available for the following programming languages: Java, C, C++ and Python. The toolkit is currently evolving into a Simple API for Grid Applications.

What is valuable from our point of view, is the concept of providing invariant, simple API for accessing grid resources and the idea of adaptors dedicated for a specific middleware technology that can can be switched during execution of application. Although GAT presents a good approach to the problem, it does not adapt to dynamic grid environment automatically at runtime. Further more, we are strongly convinced that there is a need for a more high-level solution that would enable to use services, jobs or components within applications in an object-oriented-style.

[2] Image from *http://www.gridlab.org/WorkPackages/wp-1/*

3.3.2. GEODISE

This project introduces a very interesting concept of accessing grid resources within a scripting language. Such an approach enables to use a set of control structures, thus facilitating expressing experiment logic. A set of computational toolboxes is provided that allows interfacing Condor and Globus, managing proxy certificates, job submission and file transfers. GEODISE includes toolboxes for Jythoon and Matlab. System architecture illustrated in Fig. 3.3[3].

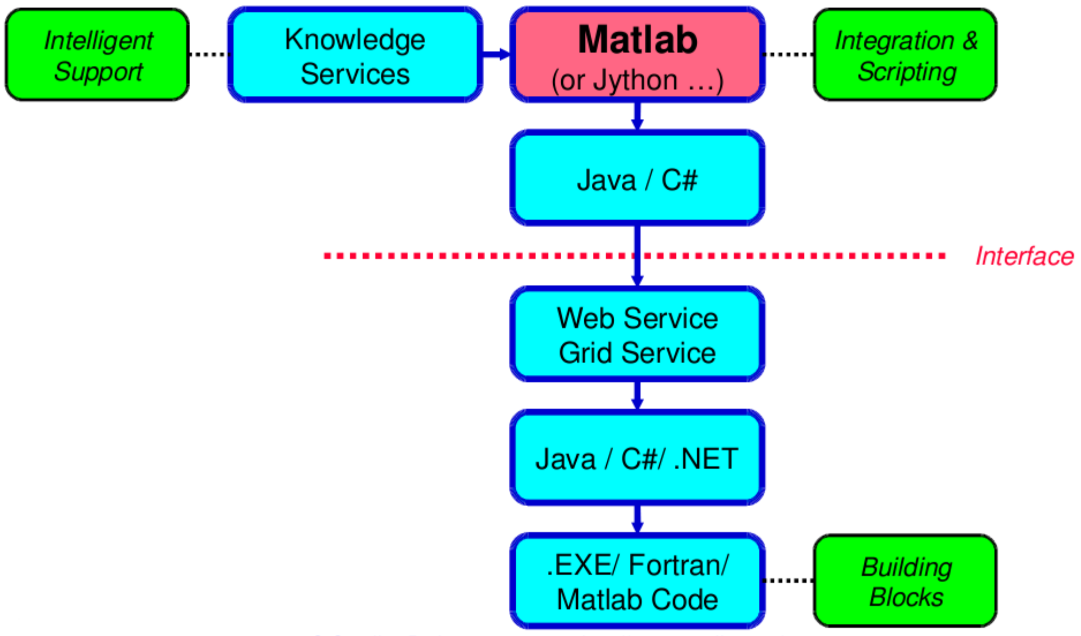


Figure 3.3. : upper-level components and services facilitating experiment development and scripting language accessing grid resources.

GEODISE is a very attractive solution but it has disadvantages that prevents it from being accepted and used in scientific communities. For example, it uses a commercial software like Matlab, Microsoft .NET or IBM WebSphere.

3.3.3. NetSolve/GridSolve

NetSolve/GridSolve [38] is a RPC-based client/agent/server system that provides users with a remote, uniform and efficient mechanism to access both hardware and software components. NetSolve is built upon standard Internet protocols such as TCP/IP. The process of accessing remote resources is transparent for the user. The client library, included in the user application, contacts the agent for a list of capable servers. Subsequently, it contacts a selected server and sends input parameters. The server executes the appropriate service and returns an output or an error status to

[3] Image from *http://www.geodise.org/files/slides_posters/workflow_COX_geodise_4Dec2003_public.ppt*

a client. Fig. 3.4 [4] illustrates the system architecture. Significant contribution

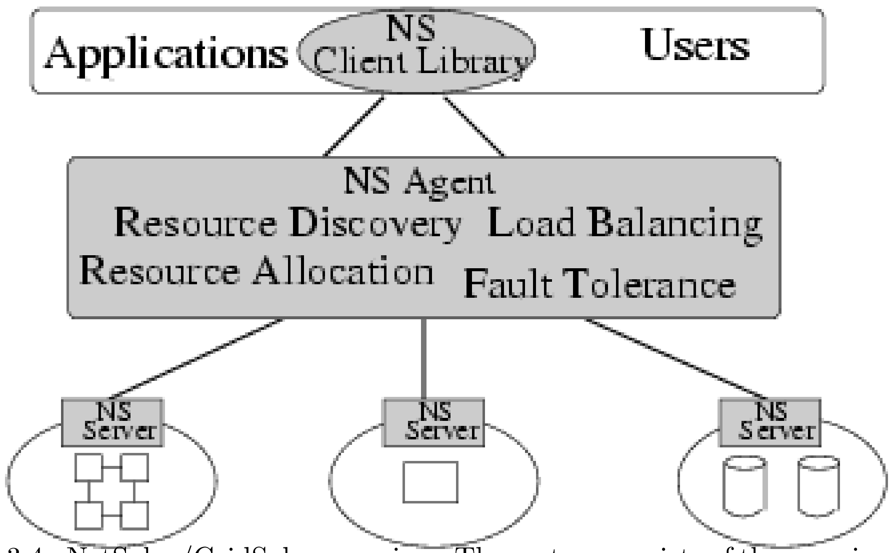


Figure 3.4. NetSolve/GridSolve overview. The system consists of three major components: client, agent and computational resources(servers).

of this project is the idea of dividing the system into three layers (client, agent and workers), that allows automated selection and transparent access to resources. This system, however, has some disadvantages. A dedicated machine for an agent is required. Moreover, developer can not access multiple technologies, nor use different programming models.

3.3.4. Web Service Invocation Framework

Web Service Invocation Framework (WSIF) [39] is an Apache project that is based on the concept of separating the API and the communication protocol. It provides a Java API for invoking services regardless of the way how they were published or their locations, provided that they are described by a WSDL. Instead of using the SOAP protocol, developers interact with an abstract representations of services through their WSDL descriptions, thus use the same programming model for all services. This framework inspects service meta-data and on this basis allows stubless or completely dynamic invocation of a service. Moreover, WSIF enables to select the binding of a service at runtime and update the implementations of a binding.

This framework provides additional binding extensions that allows describing Enterprise Java Beans (EJB), local Java classes, software accessible using Java Connector architecture and applications using Java Message Service (JMS) with WSDL documents. Thus all these technologies and programming models are normalized in terms of descriptions and can be used in a uniform manner. We believe, that such

[4] Image from *Users' Guide to NetSolve V2.0*

a feature is extremely-recommended in development of grid applications. Despite of many advantages and interesting concepts, this framework is not suitable for solution of the problem defined in Section 1.3.1, because it is more business-centric, it does not select the optimal computing resource nor support loadbalancing. Finally, WSIF project is focused on Web Services and Java technologies and does not support scientific computing or jobs.

3.4. Conclusions

Scientific communities should be able access the grid environment in a more user-friendly manner. Complicated problems can be solved by applications using computational power supplied by grids. Developers of such applications, should not be troubled with interfacing heterogeneous and dynamic environment, the process of finding and selecting appropriate resources should be transparent for them. In addition, they should be able to take advantage of their skills as well as of programming properties, which were described in section 2.1 and are offered by various middleware suites.

In spite of advantages of workflow systems, these projects have some limitations, which are present in all solutions. Due to limitations of control structures, building a workflow is not a natural way of expressing application logic, therefore more complex experiments are difficult to define. Moreover, experiment developers are limited because they can use functionality that is registered in the system. Every operation, even a trivial one, like changing data format, is executed as a single step of workflow. Developers can not use any of their own local code, nor external services. Due to these limitations, experiment developers are not able to prototype experiments. Workflow-based systems can not be extended easily because it involves administration effort.

We have analyzed existing solutions that enable to develop and execute applications accessing the grid environment. We have pointed out which concepts and approaches are profitable and listed deficiencies of discussed projects with regard to the problem defined in Chapter 1. Furthermore, we have compared systems for building grid applications (see Table 3.4) in terms of:

- supported middleware,
- supported programming languages,
- support for automatic resource selection,
- capability of combining local and remote computations,
- extendability (ability to use external computing resources and ease of adding support for new middleware technology),
- server requirement (Is a dedicated machine running one or more daemon processes or services required?),
- support for programming paradigms,
- abstraction level,
- license (Can a system be used free of charge? Is source code published and can it be reused?).

Conclusion stemming from the analysis in the field of high-level grid programming is that none of the existing systems, frameworks or a set of libraries meets all requirements. Despite of that fact, reviewing existing solutions provided inspiration for defining our approach and collecting requirements as well as constituted an anchor point for our work.

Feature	**System for building grid applications**						
	Kepler	**Triana**	**K-Wf Grid**	**GAT/SAGA**	**GEODISE**	**Net/Grid Solve**	**WSIF**
Supported middleware	WS, Globus	WS, WSRF, jobs (using GAT)	WSRF	WS, Globus, UNICORE, job-oriented middleware supported by DRMAA	WS, Globus	Globus, Condor	WS, EJB, JMS
Programming languages	MoML	graphical represen-tation of workflow	GWorkflowDL	C, C++, Python, Java	Jython, Matlab	C, Fortran, Matlab	Java
Automatic resource selection	yes	yes	yes	yes	no	yes	no
Local and remote computations	yes	yes	no	yes	yes	yes	yes
Extendable	yes	no	no	yes	no	no	yes
Requires a server	no	yes	yes	no	no	yes	no
Programming paradigm	actor -oriented	drag and drop	formal description of user request	language -dependant	functional, object -oriented / functional	Remote Proce-dure Calls	object -oriented
Abstraction level	high	high	high	low	low	low	low
License	open source	open source	open source	open source	use commercial components	free, see file COPYING in the distribution	open source

Table 3.1. Comparison of solutions allowing building grid applications.

Chapter 4

Vision of a Grid Operation Invoker

This chapter introduces our solution of the problem. First, we define a novel virtualization layer over grid environemts. Subsequently, a vision of a system providing uniform interface to grid resources within a scripting language is presented. Finally, we point out advantages of our approach over similar systems.

4.1. Abstraction over the grid environment

Development and running high-level applications on grids is still very difficult and demanding, due to heterogeneity and dynamic nature of the environment. Moreover, to meet requirements of various groups of users an abundance of middleware packages have been developed that, in most cases, are not able to interoperate. We believe that developers of an application should only be focused on the essence of the problem they are solving rather than being concerned with selecting resources and interfacing them in theirs specific protocols and hence another layer of virtualization in required. Due to this fact we introduce *Grid Object*, *Grid Operation*, *Grid Object Class*, *Grid Object Implementation* and *Grid Object Instance* concepts that allows uniform and abstract description of resources that may use a wide range of middleware suites. The hierarchy of abstraction is depicted in Fig. 4.1.

Grid Operation is a computational ability provided by a software entity deployed on a grid, which is accessed remotely. Every operation is described by a signature, which defines inputs and outputs. *Grid Object Class* is a set of remote software entities that provide exactly the same set of *Grid Operations*. All members of such a class are identical from the developer's point of view in terms of provided functionality, but may differ in technological aspects. *Grid Object Implementation* constitute a subset of *Grid Object Class*, it includes all entities from a class that are published using one middleware technology, for instance Web Service or components. Implementation realizes (implements) all *Grid Operations* of a *Grid Object Class*. On this level of abstraction middleware technology is defined albeit the address of the software entity remains unspecified. This leads us to *Grid Objects Instance*, which is a running or ready to be run software entity that has its unique address thus can be accessed through the network. *Grid Object* is a client side representative for an instance. It is created using the API provided by our system and enables developer to invoke *Grid Operations* in the same manner as methods on ordinary objects.

These concepts of *Grid Operation*, *Grid Object* and *Grid Object Class* correspond respectively to a *method*, an *object* and a *class* concepts in the object-oriented programming.

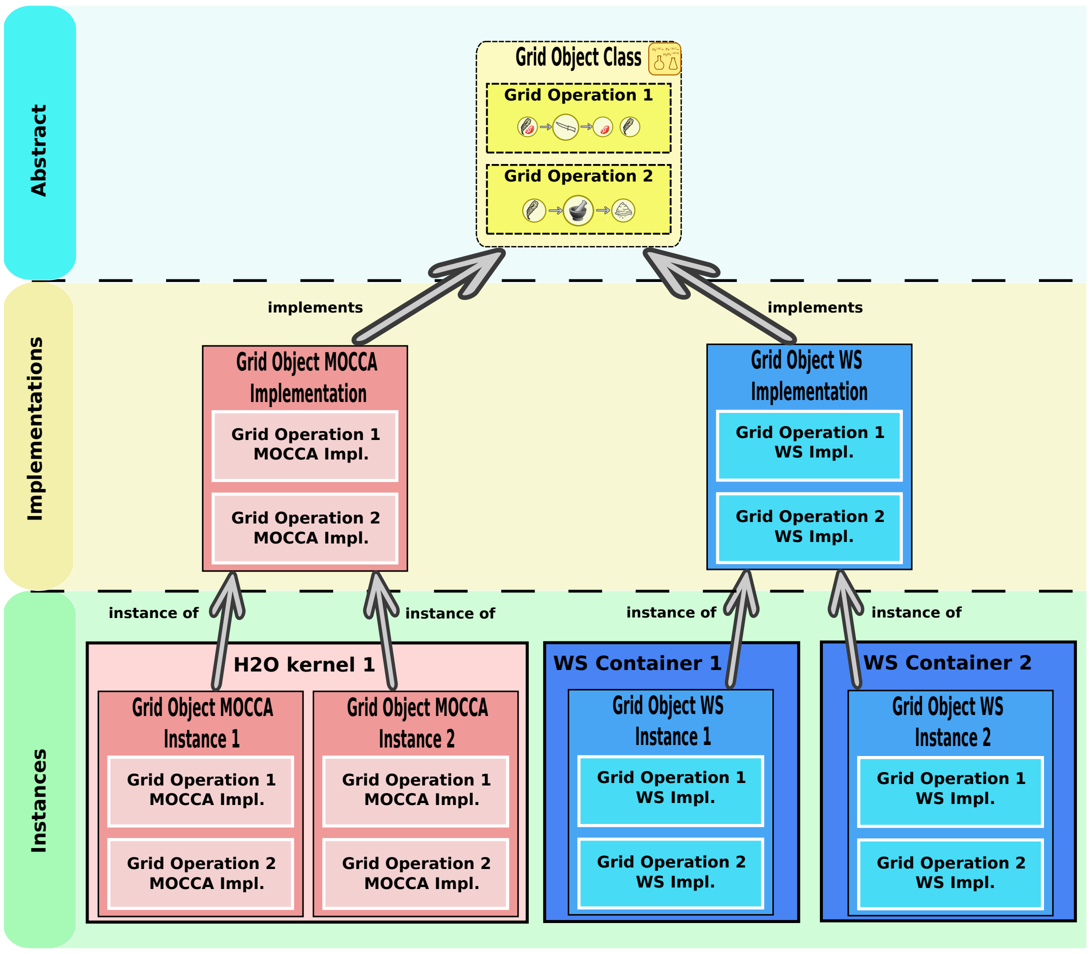

Abstract
Implementations
Instances
Grid Object Class
Grid Operation 1
Grid Operation 2
implements
implements
Grid Object MOCCA Implementation
Grid Operation 1 MOCCA Impl.
Grid Operation 2 MOCCA Impl.
Grid Object WS Implementation
Grid Operation 1 WS Impl.
Grid Operation 2 WS Impl.
instance of
instance of
instance of
instance of
H2O kernel 1
Grid Object MOCCA Instance 1
Grid Object MOCCA Instance 2
WS Container 1
Grid Object WS Instance 1
WS Container 2
Grid Object WS Instance 2

Figure 4.1. Three layers of abstraction over the grid environment: 1. Abstract layer, which contains *Grid Object Classes* defined by the set of operations they provide; 2. Implementations layer gathering entities that implement the functionality of a specific *Grid Object Class* and are categorized on the basis of the used middleware technology; 3. Instances layer collects *Grid Object Instances*, which are running or ready to be run on user's demand implementations.

In spite of the abstraction over the grid, programming properties of *Grid Object Instances* should be exposed to developers to enable them to implement more efficient applications.

4.2. Development of grid applications

Developer should be able to work both on the highest and lowest levels of abstraction. In the former case, only the required functionality should be specified by giving the name of the *Grid Object Class*. Finding an appropriate instance capable to perform the computation and selecting the optimal computing resource should be done automatically and be a transparent process for the user. In the latter case, developer should either provide an unique identification of the resources or technology information describing the instance.

We believe that using grids from within a scripting language should be as simple and concise as the example code presented in Fig. 4.2. Developer should only require one factory class in a script, that will provide a uniform interface for creating *Grid Objects*. After creating such an object, it should be used in an ordinary manner. Our main objective is to allow development of high-level applications in such a way. A Grid Operation Invoker (GOI) is a universal grid client that is, in our opinion, a

```
require 'cyfronet/gridspace/goi/core/g_obj'

classifier = GObj.create('weka.OneRuleClassifier')
classification = classifier.classify(data)
```

Figure 4.2. Invoking an operation on a grid from a script using Grid Operation Invoker API.

solution of the problem defined in Chapter 1. It is a light-weight, client-side library that allows uniform and transparent usage of *Grid Object Instances* within a script. A simple, yet fully functional, API is provided to create representatives on both levels of abstraction. Developers can use *Grid Objects* within the script in the same manner as ordinary objects, regardless of the instance's underlying middleware technology. The system can choose the instance to be used and take over the communication with it, thus endusers avoid obstacles associated with finding the optimal computing resource and interfacing it in its specific protocol. This effort is moved from users to our software system.

4.3. New features provided by the Grid Operation Invoker

In our opinion our system is significantly different from similar solutions (see Chapter 3). We believe that our approach has many advantages, among which the most important are:

- using a scripting language allows full expressiveness in terms of application logic combined with easy to grasp language syntax and clear and concise code;

- our system is light-weight: its size is small, it does not introduce any dependencies except Java Virtual Machine and a script interpreter, it does not start any server or daemon processes and it can be used on every modern personal computer;
- ease of customization to individual needs, ability to cooperate with a diversity of registries, optimizers or to operate as a standalone system
- client-side system – does not require any external infrastructure or administration;
- ease of installation;
- based on reliable technologies and standardized protocols;
- open source licensee;
- ability to extend the system to support emerging middleware suites;
- universal tool that can be applied for solving problems in a variety of domains;
- can be used as a backend for execution engines.

4.4. Summary

In this chapter we have presented the concept of Grid Object abstraction over the grid. It enables to describe resources on various level of abstraction. On the highest level, we define only the provided functionality. On the implementation level we add information about the used middleware technology. Finally, on the lowest level we provide full data that enable to invoke remote operations. Subsequently, we have introduced the Grid Operation Invoker, which realizes the concept of three-level abstraction and provides uniform and transparent access to grid resources. Finally, we have listed advantages of our system.

Chapter 5

Analysis phase of the Grid Operation Invoker

Having reviewed middleware suites and high-level systems we now present the analysis of the Grid Operation Invoker system. We start with an overview of the system, which we follow with use cases description. Finally, we list system requirements divided into functional and nonfunctional ones.

5.1. Overview of the Grid Operation Invoker system

The Grid Operation Invoker main objective is to facilitate developing and running high-level applications accessing grid resources. Creating such high-level applications should be as easy as implementing software that is executed locally. Moreover, running high-level applications should not differ from executing ordinary applications. The system should allow uniform and transparent access to the grid environment from within a scripting language. It should be able to delegate execution of a *Grid Operation* to a specific, optimal, selected at run-time *Grid Object Instance* using communication protocol specif for the instance. The GOI system should support all leading middleware suites and be easily extendable.

The system should be implemented as a client-side, light-weight library for a scripting language, extending it with the capability of interfacing the grid environment. The scripting language being enhanced should be broadly accepted and well documented. It should be easy to grasp, powerful object-oriented language with a clear and concise syntax. The GOI system should be easy to install and customize for endusers. The idea of the Grid Operation Invoker system is depicted in the Fig. 5.1.

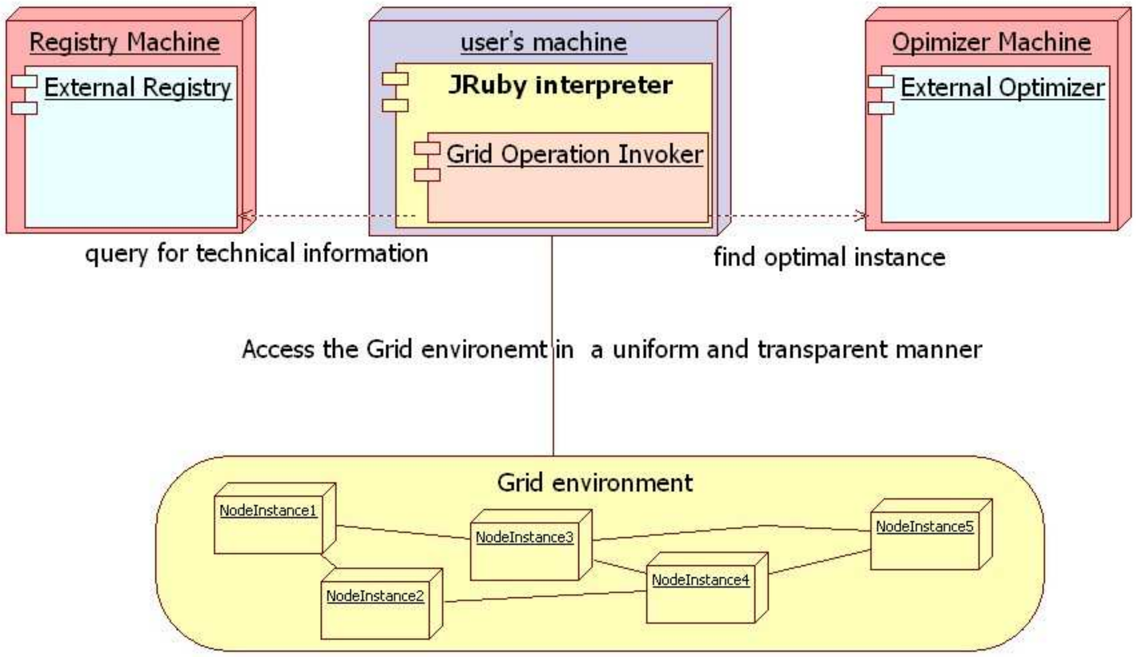


Figure 5.1. Overview of the Grid Operation system.

The optimizer and the registry presented in Fig. 5.1 are out of the scope of this work, nevertheless the GOI system should be able to cooperate with various external optimizers and registries. Furthermore, the system should provide simple local implementations of a registry and an optimizer. The need for an optimizer and for a registry have been identified in Section 1.3.3 and is also discussed in Section 5.4.

There are two kinds of users of our system: developers of high-level applications and scientists using these applications to conduct research. The former group is provided with a simple and uniform API enabling them to use grid resources in a coherent and transparent way. They can use abstraction over the grid, which we have introduced in Section 4.1. Developers should be able to create *Grid Objects*, which represent remote software entities, and use them in the same manner as ordinary objects. They should be able to specify the requested functionality by providing the *Grid Object Class*, or choose a specific *Grid Object Instance*.Therefore, they can focus on solving the problem rather then being overwhelmed by obstacles associated with accessing remote resources. Besides that, developers should be aware of the programming properties of underlying middleware technologies (see Section 2.1) in order to take advantage of their programming skills. Scientists can easily execute experiments that automatically and seamlessly adapts to the dynamic grid environment, because optimal resources are selected during run-time.

5.2. Use cases of the Grid Operation Invoker system

In order to define the exact functionality of the Grid Operation Invoker we describe use cases of the system. In Section 5.1 two kinds of users have been identified: developers of a high-level grid application and a scientists who execute the application (experiment) to produce results for their research. We present use cases with regard to this classification of users. We provide UML use case diagrams for each use case and discuss them in more details.

5.2.1. Development of a high-level grid application

Developers need to solve a highly-complicated scientific problem. In order to achieve this goal, they need to use functionality, computational power and storage provided by distributed and heterogeneous resources of the grid environment. Developers should focus on the problem and be able to access the dynamic grid environment in a uniform and transparent manner. The most natural and accepted approach is to use an object-oriented programming techniques and to use *Grid Objects* (see Section 4.1) representing remote software entities (*Grid Object Instances*). Fig. 5.3 illustrates developing with the GOI library.

Initially, developers needs to include the *GObj* class providing uniform interface for creating *Grid Objects*(see line 1 of Fig. 5.2).

```
1 require 'cyfronet/gridspace/goi/core/g_obj'
2
3 classifier1 = GObj.create('weka.OneRuleClassifier')
4 classifier2 = GObj.create_instance(7)
5 classifier3 = Resource.new(techInfo)
```

Figure 5.2. Development of grid application using the GOI library. Lines: 1.) Including the *GObj* class. 3.) Creating a *Grid Object* of a given *Grid Object Class*. 4.) Create a *Grid Object* for a specific *Grid Object Instance*. 5.) Create a *Grid Object* using low-level API of *Resource* class.

From now on, developers can use a simple API provided by the GOI library enabling them to create *Grid Objects* in three ways:

1. Specify only the required functionality by providing the name of the *Grid Object Class*. This is the most abstract method of creating a *Grid Object*. This case is illustrated by line 3 in Fig. 5.2.
2. Select a specific instance by providing a unique identifier of an instance. This assumes that the developer is absolutely sure that the identifier points to the instance that he/she wishes to use. This case is presented by line 4 in Fig. 5.2.
3. Use low level API. In such a case, developer needs to require an appropriate resource class, which is used to interface the *Grid Object Instance* in its specific protocol, and to provide technology specific data(for more information on technology information please refer to the Appendix D). This case is shown by the line 5 in Fig. 5.2.

Having created a *Grid Object* developers can use it in the same manner as ordinary objects. Invocations of *Grid Operations* will be delegated to *Grid Object Instances* during the run-time seamlessly. Developers can implement the application logic using full capabilities of the modern object-oriented scripting language.

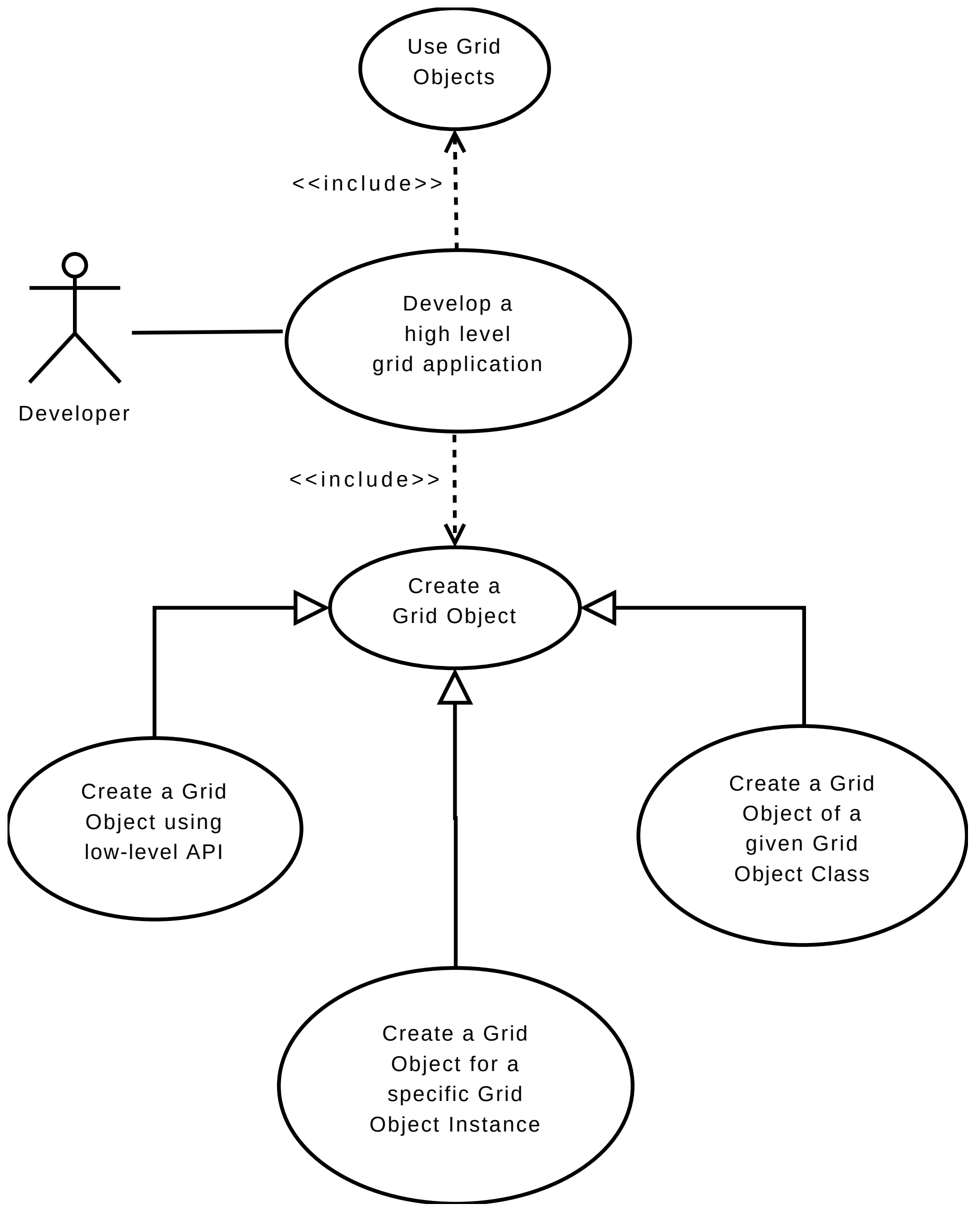

Use Grid Objects
<<include>>
Developer
Develop a high level grid application
<<include>>
Create a Grid Object
Create a Grid Object using low-level API
Create a Grid Object of a given Grid Object Class
Create a Grid Object for a specific Grid Object Instance

Figure 5.3. Use case diagram 1: Developer implements a high-level grid application.

5.2.2. Executing a high-level grid application

Once a grid application is developed, scientist can use it to conduct their research. They can execute these applications likewise normal scripts. The process of invocations of *Grid Operations* is absolutely transparent from the user's point of view. During the run-time, *Grid Object Instances* are selected automatically or are given explicit in the script. In both cases user is not concerned with finding, selecting and interfacing directly grid resources. During an execution of a script, *Grid Operations* can be invoked either in stateless or stateful manner. Moreover synchronous or asynchronous calls can be used. These decision are made by the developer while implementing an application. Besides that, running application can manage components (see Section 2.2.2). Components can be deployed and then they can be, but do not have to be destroyed. Furthermore, previously deployed components can be destroyed.

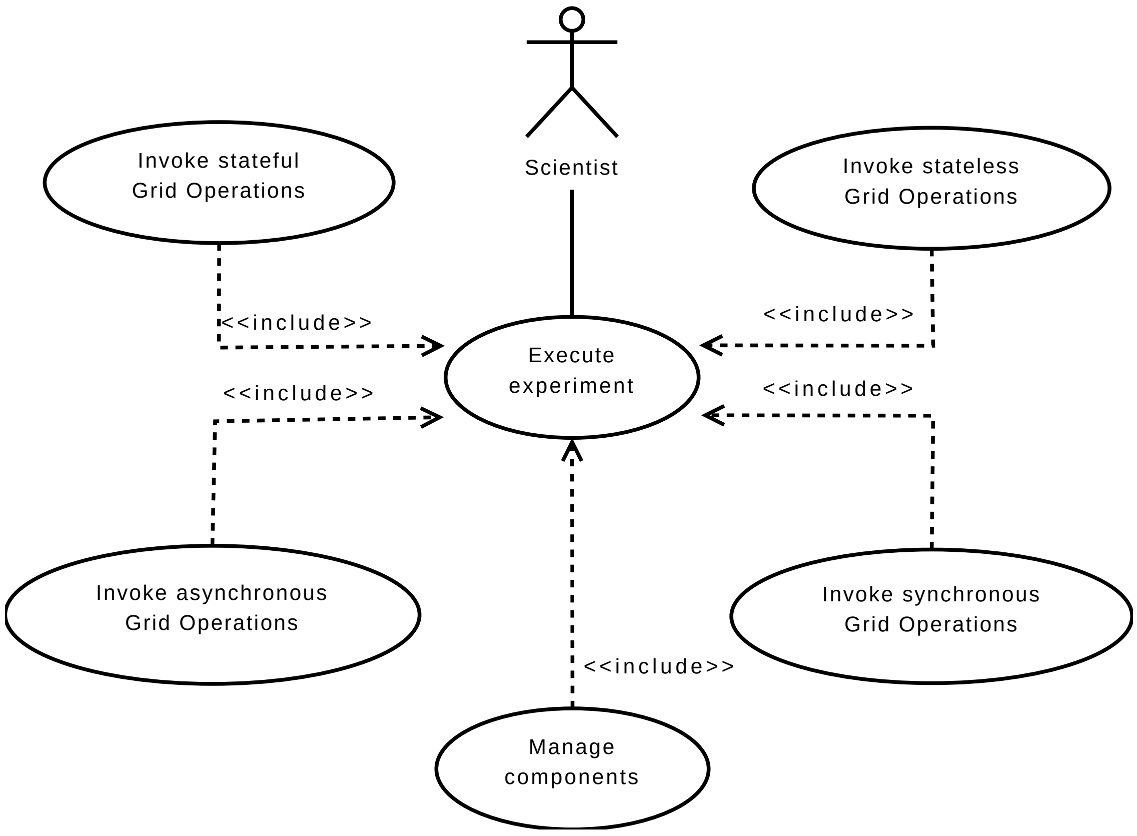

Scientist
Invoke stateful Grid Operations
Invoke stateless Grid Operations
<<include>>
<<include>>
Execute experiment
<<include>>
<<include>>
Invoke asynchronous Grid Operations
Invoke synchronous Grid Operations
<<include>>
Manage components

Figure 5.4. Use case diagram 2: Scientist executes an experiment (a high-level grid application).

5.3. Requirements

In this section we gather all requirements for the Grid Operation Invoker system based on overview of the GOI system (Section 5.1) and discussion on use cases (Section 5.2). We divide them into functional and nonfunctional ones.

5.3.1. Functional requirements

Here we describe functional aspects of the Grid Operation Invoker system. We define and list functional requirements:

1. Enable coherent and transparent usage of grid resources. In order to harness the grid environment a standardized interface for executing operations over the grid is needed. Moreover, the process of finding optimal resources and interfacing them in their specific protocols should be transparent for the application developer and enduser who executes the experiment.
2. Provide an API enabling to work both on high and low level of abstraction. Developer should be not only able to work on the *Grid Object Class* level of abstraction, but also should be able to choose a specific instance, due to software quality, reliability factors or accounting issues.
3. Provide a full set of control structures. Our system should allow full expressiveness in terms of application logic.
4. Add programming features (see Section 2.1) such as asynchronous call, concurrent execution or support for the object-oriented programming. These features facilitate development of efficient applications and enable developers to employ theirs programming skills.
5. Handle required data conversions between various technologies. Neither developers, nor experiment users should be concerned about diverse data formats used by underlying middleware technologies. If a conversion is required it should be done automatically and seamlessly by the system.
6. Facilitate experiment reuse and support nested experiments, because highly complicated problems may require decomposition into smaller parts. This can be achieved by supporting object-oriented paradigm.

5.3.2. Nonfunctional requirements

In this part of the thesis, we focus on nonfunctional qualities of the Grid Operation Invoker. The nonfunctional requirements of the system are as follows:

1. The system should be based on a proven and reliable technologies and standardized protocols.
2. The GOI should be a light-weight, client-side library.
3. Experiment notation should be based on a common scripting language with clear and concise code. The language combined with our library should be easy to grasp and use.
4. The system should provide a convenient mechanism for adding support for emerging middleware technologies.

5. The distribution of the system should be a single package that is easy to install even for a user with little computer skills. The system should be platform independent.
6. The Grid Operation Invoker should be able to work as a standalone system.
7. The system should be customizeable to use various external optimizers and registries.
8. The implemented system should be a general use solution rather than being a domain specific one.
9. Ability to serve as a backend for a diversity of upper level systems.

5.4. Dependencies

The Grid Operation Invoker requires two components:

1. an optimizer that can select the optimal *Grid Object Instance* and return a its unique identifier,
2. a registry that stores technology information (see Appendix D) about instances and can provide this information for an instance selected by the optimizer.

Our system can work as a standalone system and use simple local implementations of optimizer and registry, although it is adviced to use external optimizer and registry. In the latter case, the system would become more scalable and enable to build communities of scientists and share services and resources. We have defined interfaces that needs to be implemented by an optimizer and a registry in order to serve for the Grid Operation Invoker.

The system main objective is to provide access to grid resources, therefore it depends on the underlying middleware technologies it supports. The technologies we intend to support are as follows:

- Web Service,
- WSRF,
- WTS,
- LCG (EGEE),
- MOCCA.

5.5. Summary

In this chapter we have presented the analysis phase of the Grid Operation Invoker. We have provided overview of the system, discussed use cases, collected requirements and presented system dependencies. Generally speaking, the Grid Operation Invoker should be a client-side, light-weight library that extends the scripting language with capability of interfacing the grid environment in a uniform and transparent manner. The system should use an external optimizer that can select an optimal *Grid Object Instance* and a registry that can provide technology information about selected instance.

Chapter 6

Design and implementation of the Grid Operation Invoker

This chapter describes the design and implementation of the Grid Operation Invoker. First, we chose the implementation technology, which has implications on the whole design. Subsequently, we introduce the architecture and proceed with GOI algorithm. Next, we provide a more detailed explanation of the system structure by describing packages, interfaces and classes. Subsequently, we provide sequence diagrams that present system use cases. Finally, we discuss provided and required interfaces of the system.

6.1. Implementation technology

We decided to invert the classic order of stages of the design process and to choose the implementation technology at the beginning. Capabilities of the selected technology have significant impact on the design and implementation of the system. In Section 4.2, we stated that our system is a library enhancing the scripting language with ability to access grid resources, therefore we narrowed the diversity of considered technologies to the most accepted scripting languages which, in our opinion, are Ruby, Perl and Python. Criterion that we have applied to these languages are the following:

- **Ease of use**. Is the syntax of the language clear and the code concise? Is it a high-level language?
- **Support for the object-oriented paradigm**. Does the language allow using object-oriented programming techniques?

- **Support for distributed computing**. Does the language allow distributed computing? Does it provide native support for any of middleware technologies that are in scope of our interest?
- **Metaprogramming**. Does the language enable to dynamically generate and then execute source code?
- **Documentation**. Is the language well documented? Are there additional materials on the Web and publication on it?
- **Platform independence**. Do interpreters for all platforms exist? Is the code portable?
- **Language popularity**. Is the language broadly accepted? Does it gain popularity? Is it evolving and will it be supported in future?

Having analyzed these requirements we have found the Ruby language to be the best choice. It is a modern, object-oriented and powerful programming language with easy to understand code. Ruby is a mature solution with a wide range of multipurpose libraries available and built-in support for Web Service technology. A multitude of publications for developers is available. Ruby is a dynamic, interpreted language and allows metaprogramming. It has a Java implementation, JRuby, therefore it is platform independent. What is even more significant from our point of view, is the fact that JRuby enables to use Java objects within the script, thus Java client-side libraries for accessing middleware suites can be reused.

Ruby supports object-oriented programming, albeit it is not a typed language. It uses the *duck typing* paradigm (see Chapter 23 of [40]). While interpreting the script the interpreter make sure that an object responds to a given method, not if the type (class) of the object is appropriate. In spite of this fact, we use UML diagrams in the following sections of this chapter to express various aspects of system design.

6.2. Structure of the Grid Operation Invoker

The architecture of the Grid Operation Invoker is a result not only of the state-of-the-art study in fields of middleware suites and high-level systems (see Chapters 2 and 3), but also reflects our approach to the problem (see Chapter 4) as well as the analysis phase (see Chapter 5). Fig. 6.1 [1] presents the architecture of the Grid Operation Invoker.

The GOI system has modular architecture. It is built of components with well-defined interfaces to allow reusability and interoperability with external systems. The main component provides the high-level *GObj* interface for creating *Grid Objects*. *Optimizer Client* is used to delegate the query for an optimal *Grid Object Instance* to the actual Optimizer. The *Registry Client* has a very similar role. It delegates the queries to a Registry that provides technology information about *Grid Object Instances*. *Optimizer Client* and *Registry Client* can be replaced by another components that provide the same interfaces but can use different Optimizers and

[1] Image from [41]

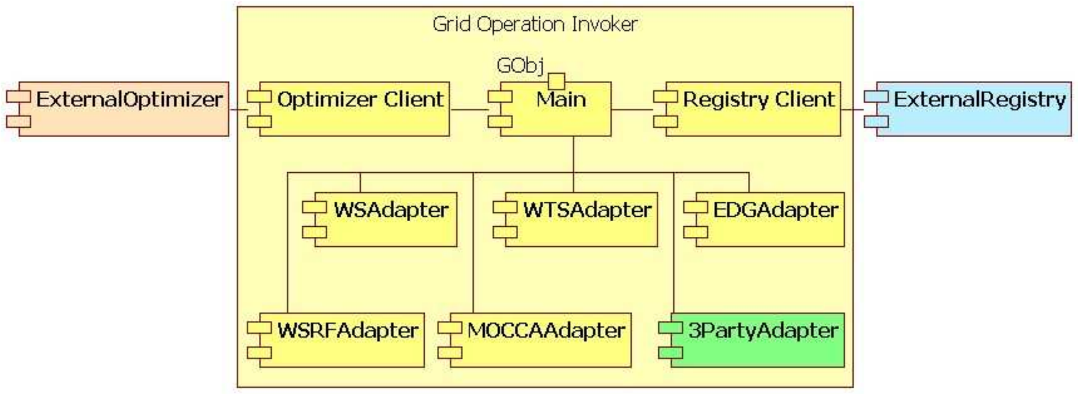


Figure 6.1. Grid Operation Invoker architecture.

Registries, thus the Grid Operation Invoker can be customized and cooperate with a wide range of externals components. For more detailed information on the GOI customization please refer to Section A.5. Adapters are used to produce *Grid Objects* capable to interface various middleware suites in their specific protocols. There is one dedicated adapter for each supported technology.

6.3. Algorithm of the Grid Operation Invoker

The support for multiple middleware technologies and the ability to select the *Grid Object Instance* at a run-time is based on the concept of adapters. An adapter is dedicated for a specific middleware technology and is loaded at a runtime. The names of the adapters are generated at runtime on the basis of technology information for an instance, thus the system can be easily extended to support emerging technologies just by placing an adapter files in the appropriate directory. For information regarding extending the GOI system please refer to Appendix C.

There can be distinguished two phases in the operation of the Grid Operation Invoker. The first one is the process of creating a *Grid Object* and the second one is invocation of remote operations. The former phase is depicted in Fig. 6.2.

The process of creating a *Grid Object* is divided into three stages:

1. Querying an optimizer for an optimal *Grid Object Instance* of a given *Grid Object Class*. The selection of the optimal instance is performed either in a simple local optimizer or is external to the GOI system.
2. Querying a Registry for a technical information for a given *Grid Object Instance*, determining the adapter class for the selected instance and loading needed adapter if it was not loaded earlier. Again, the Registry can be either a simple local implementation or a standalone external system.
3. Producing a *Grid Object* using a dedicated adapter.

As stated in Section 5.2.1, developers can create *Grid Objects* in three ways.

If the most high-level approach is taken and the name of the *Grid Object Class* is provided as a parameter, all three stages described above are involved. If developers choose an *Grid Object Instance* they wish to use, they must provide the identifier of the instance, therefore the first stage is omitted. In case of using low-level API, developers provide technology information and only the last stage is executed.

The second phase of the GOI operation is using *Grid Objects* within the script to invoke *Grid Operations*. From the developer's point of view, *Grid Object* is used in an ordinary manner, nevertheless invocations of remote operation are transparently delegates to *Grid Object Instances* and results are returned. If any data conversions are required, they are done automatically without user attention.

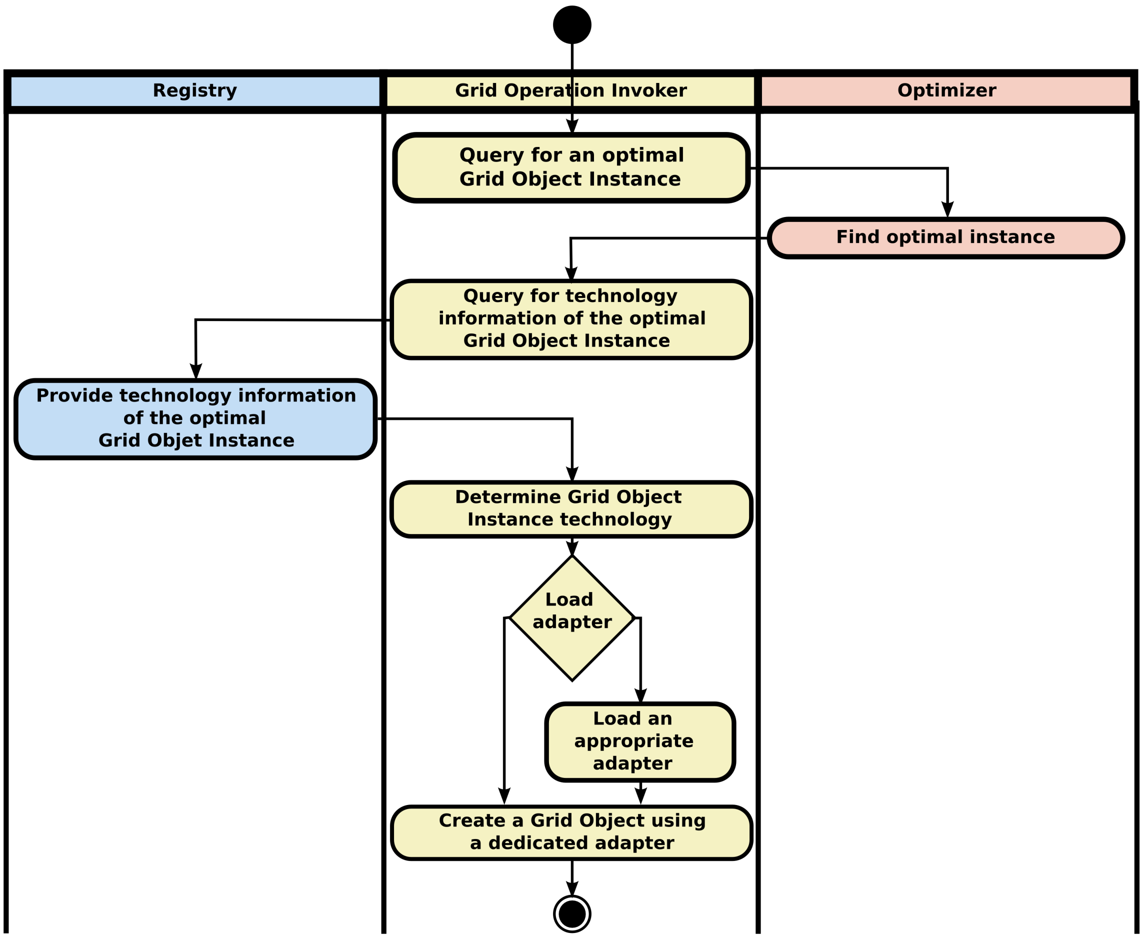

Registry
Grid Operation Invoker
Optimizer
Query for an optimal Grid Object Instance
Find optimal instance
Query for technology information of the optimal Grid Object Instance
Provide technology information of the optimal Grid Objet Instance
Determine Grid Object Instance technology
Load adapter
Load an appropriate adapter
Create a Grid Object using a dedicated adapter

Figure 6.2. Grid Operation Invoker activity diagram for the process of creating a *Grid Object* using an external Optimizer and an external Registry. Stage 1: Querying an Optimizer for an optimal *Grid Object Instance* of a given *Grid Object Class*. Stage 2: Querying a Registry for technology information for a given *Grid Object Instance*, determining the adapter class and loading a dedicated adapter. Stage 3: Creating a *Grid Object* using a dedicated adapter.

6.4. Detailed design of the Grid Operation Invoker system

In Section 6.2 we have provided an overview of the system by describing GOI architecture. In this section we describe the structure of the system with more details. Fig. 6.3 presents main packages, classes of the system and relations between them. The system is divided into four packages: *core*, *adapters*, *utils* and *exceptions*. The first package provides the main functionality of the system. It implements the GOI algorithm described in Section 6.3. The *adapters* package contains classes that are dedicated to specific middleware packages and are responsible for interfacing *Grid Object Instances* in their specific protocols. Next package includes all additional utilities classes that are required by the system. The last package is not presented on the class diagram, nor discussed in Subsection 6.4.1, because it contains only exception classes and as such does not help to understand the structure of the Grid Operation Invoker.

6.4.1. Description of the Grid Operation Invoker packages

The *core* package consists of:

- *GObj* class. It is an abstract factory that provides high-level API for creating *Grid Objects*. It implements two class methods *create* and *create_instance* that return the same result (for information on class methods in Ruby please refer to Chapter 3 of [40]). The former method enables developers to produce *Grid Objects* of a given *Grid Object Class*, which name is passed as a string value. The latter method accepts a long integer that is a unique identification of the *Grid Object Instance*, thus allows producing representative for a given instance. *GObj* is the main class of the system and if developers do not wish to use low-level API, it is the only class that needs to be required. This class uses Registry and Optimizer Client classes to query for an optimal instance and technology information of an instance. Due to being an abstract factory, this class does not produce *Grid Objects* directly, but it loads appropriate dedicated adapter and delegates the creation of representative to it.
- *Constants* class. This class contains all constants that are used by the system. These constants can have fixed values assigned or be evaluated at a runtime. For instance, this class parametrizes the *GObj* class, it specifies the names of Optimizer and Registry Client classes that are used. For more information on constants used in the GOI system please refer to Section A.5.
- *OptimizerClient* and *RegistryClient* are analogous classes. The former is responsible for delegating queries for an optimal instance of a given *Grid Object Class*. It must provide the *find_optimal_instance* method which accepts a name of the class as a string and return a unique identifier of the instance. The latter interfaces the registry. It delegates the query for technology information to a registry. These classes are responsible for any data conversions that are necessary between the GOI data format and formats used by optimizers and registries. These classes can be replaced and thus different optimizers and registries can be used by the GOI system.

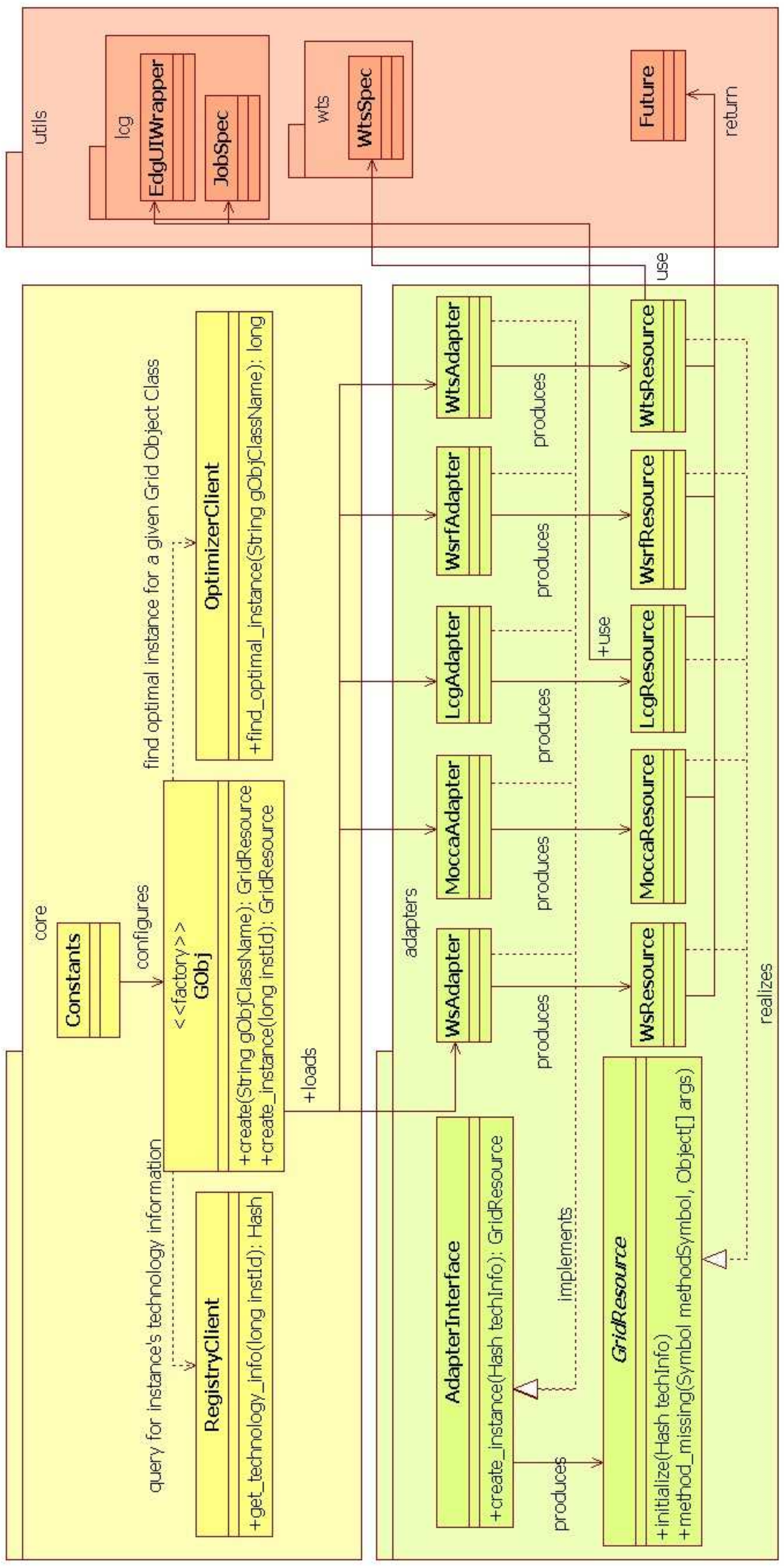

utils
lcg
EdgUIWrapper
JobSpec
wts
WtsSpec
Future
return
use
core
Constants
configures
<<factory>>
GObj
+create(String gObjClassName): GridResource
+create_instance(long instId): GridResource
find optimal instance for a given Grid Object Class
OptimizerClient
+find_optimal_instance(String gObjClassName): long
query for instance's technology information
RegistryClient
+get_technology_info(long instId): Hash
+loads
adapters
WsAdapter
MoccaAdapter
LcgAdapter
WsrfAdapter
WtsAdapter
produces
produces
produces
produces
produces
WsResource
MoccaResource
LcgResource
WsrfResource
WtsResource
+use
AdapterInterface
+create_instance(Hash techInfo): GridResource
implements
produces
GridResource
+initialize(Hash techInfo)
+method_missing(Symbol methodSymbol, Object[] args)
realizes

Figure 6.3. Grid Operation Invoker class diagram.

The *adapters* package includes the following classes:

- *AdapterInterface.* It defines the interface of the class that is responsible for producing *Grid Objects* of one specific *Grid Object Implementation* (instances remotely exposed by one specific middleware technology). Realizations of this class are loaded at a runtime by the *GObj* class.
- *GridResource.* This class is an abstract class that represents a generic computational resource of the grid environment. It can not be instantiated and used within the script, since it can not represent a *Grid Object Instance* of any middleware technology.
- Realizations of *AdapterInterface.* These class, each dedicated for one specific middleware technology, are capable of producing a *Grid Object* for one middleware technology.
- Classes that extend the *GridResource* class and are specific for one middleware technology. These classes are proxies, that provides the functionality of remote software within the script.

We have implemented adapters for the following middleware suites:

- Web Services,
- WSRF,
- MOCCA,
- LCG,
- Witty Services.

These adapters enables to use a diversity of middleware packages in a uniform manner. An experiment at a run-time may adapt to the dynamic environment and every time it is executed it can use optimal *Grid Object Instances* selected at a run-time by an *Optimizer.*

Web Service

An adapter for this technology is based on the Ruby built-in support for this technology. It produces *Grid Objects* of *WsResource* class. Web Service is an accepted standard, although there may be some differences between services providing the same functionality. First of all a service may, but do not have to be, described by a WSDL. Next, it can use either a RPC or DOCUMENT binding style. The former binding accepts a list of arguments, while the latter accepts a data in a structure defined in the WSDL of the service. Such a structure is generated by the *WsResource* at a runtime, thus services using various styles are used in a coherent manner. Moreover, services that do not have a WSDL are also supported.

WSRF

An adapter for this techology is based on the Globus 4 Java implementation of the *WSRF* specification. Required *Java* classes and stubs are included at a runtime. *WsrfResource* object encapsulates a *Java* proxy object that enables to invoke operations on stateful services. Both the *single resource* and *multiple resource* services are supported.

MOCCA

Support for this component framework is achieved by reusing its Java client-side library. *MoccaResource* class enables also to create and destroy components, as well as manage their connections. While invoking a method of a MOCCA component there are some additional information required, for instance the name of the component's port that is used. All necessary information is extracted from technology information and added to a call of MOCCA component method automatically by the *Grid Object* of *MoccaResource* class.

LCG

EDG User Interface command line executables are wrapped by a Ruby class and used through it to submit and manage jobs. When a *Grid Operation* is invoked on an LCG *Grid Object Instance*, a JDL file is generated and a job is submitted. Then the status of the job is periodically check. The output files or error file are downloaded when the job is finished.

Witty Services (WTS)

An adapter for his technology reuses a Java client-side library. *Grid Object* of *WtsResource* class use technology information to generate a specification, which determines if the inputs and outputs are transferred as files or bytes. Once a specification is ready, a job can be submitted and upon completion results downloaded.

LCG and WTS technologies are job-oriented middleware packages. They enable to access software that does not provide any remote API and does not support RPC operation invocations. Such applications, exposed via job-oriented technologies, require specific data conversions (from Ruby objects to files and vice versa). For these reasons additional wrapper classes, which are dedicated for one specific application, are used. These classes are stored as a part of technology information describing *Grid Object Instance.* Wrappers deliver the functionality of jobs in a object-oriented style and are used to produce *Grid Objects.*

The *utils* package consists of:

- *Future* class. It is used to add asynchronous invocations of synchronous methods. Asynchronous call does not block until the result is ready, but return a promise (future variable) for the actual data. Interpretation of the script will be blocked only when the actual data is needed but is not ready yet. Future variables enable to develop more efficient experiments and allow concurrency.
- *lcg* package that consists of *EdgUIWrapper* and *JobSpec* classes. The former class wraps the EDG UI commands, while the latter enables to create a description of a job, which is then used to generate a JDL file.
- *wts* package that contains *WtsSpec* class. This specification determines how the inputs and outputs are transferred, which server is used etc.

The *utils.lcg* and *utils.wts* packages are used by developers who implement wrapper classes for applications that access *Grid Object Instances* published through job-oriented technologies. For more information on implementing wrapper classes please refer to Section C.2.1. Developers of experiments are provided with higher abstraction, although can use these low-level packages.

6.5. Patterns used in the design of the Grid Operation Invoker

In the design of the Grid Operation Invoker system we used the following design patterns (all design patterns are from [42]):

- *Abstract Factory* for *GObj* class,
- *Adaptor* for adapter classes,
- *Proxy* for resource classes.

Using these best practices facilitated the design of the GOI system as well as improved the quality of the software.

6.6. Sequence diagrams

In Section 5.2 we have defined Grid Operation Invoker use cases, which can be divided into two phases: development and execution of experiments. In the former one, the GOI API is used by developers to implement experiments. The latter phase is the one in which the system works. During script interpretation the Grid Operation Invoker creates *Grid Objects* and handles invocations of *Grid Operations*. This section explains the data and steering flow for most of use cases defined in Section 5.2. This includes:

- creating a *Grid Object* for a given *Grid Object Class* (see Subsection 6.6.1),
- creating a *Grid Object* for a specific *Grid Object Instance* (see Subsection 6.6.2),
- invoking a synchronous *Grid Operation* (see Subsection 6.6.3),
- invoking an asynchronous *Grid Operation* (see Subsection 6.6.3).

Usage of low-level API to create a *Grid Object* is not depicted on a separate sequence diagram because it involves calling only the constructor of the appropriate resource class and is presented as a step number 11 in Fig. 6.4. Managing components use case (see Fig. 5.4) is not discussed, because it involves interaction with *Grid Object* of *MoccaResource* class, which is not different from invoking *Grid Operations*. *Grid Object* for MOCCA resource expose component management functionality provided by the Java client-side library. Moreover, this use case is specific for MOCCA framework and does not help to understand the Grid Operation Invoker system. Invoking a stateless and a stateful operation are not distinguished, due to the Grid Operation Invoker working the same for these two situations. It is the responsibility of a developer to be aware of the interaction mode.

6.6.1. Creating a *Grid Object* of a given *Grid Object Class*

Sequence diagram presented in Fig. 6.4 illustrates the process of creating a *Grid Object* for a given *Grid Object Class*. This process involves all three stages of the GOI algorithm (see Section 6.3).

Description of steps:

1. *create* class method of *GObj* is used to request creation of a *Grid Object* for a *Grid Object Class* which name is passed as a *String* argument.

2. An *OptimizerClient* is queried for an optimal *Grid Object Instance* for a given *Grid Object Class*.
3. The query is delegated by the *OptimizerClient* to an *Optimizer*.
4. An identifier of an optimal instance is returned to the *OptimizerClient*.
5. The identifier is forwarded to the *create* method body.
6. A *RegistryClient* is queried for technology information describing the optimal *Grid Object Instance*.
7. The query is delegated to a *Registry*.
8. Technology information for the optimal instance is returned to the *RegistryClient*.
9. Technology information is forwarded to the *create* method body.
10. On the basis of the technology information a dedicated adapter is determined and loaded. Technology adapter is requested to create a *Grid Object* for the *Grid Object Instance* described with the technology information.
11. A *Grid Object* of an appropriate resource class is created.
12. The *Grid Object* is returned to the *create* method body.
13. The *create* method is finished and the *Grid Object* is returned.

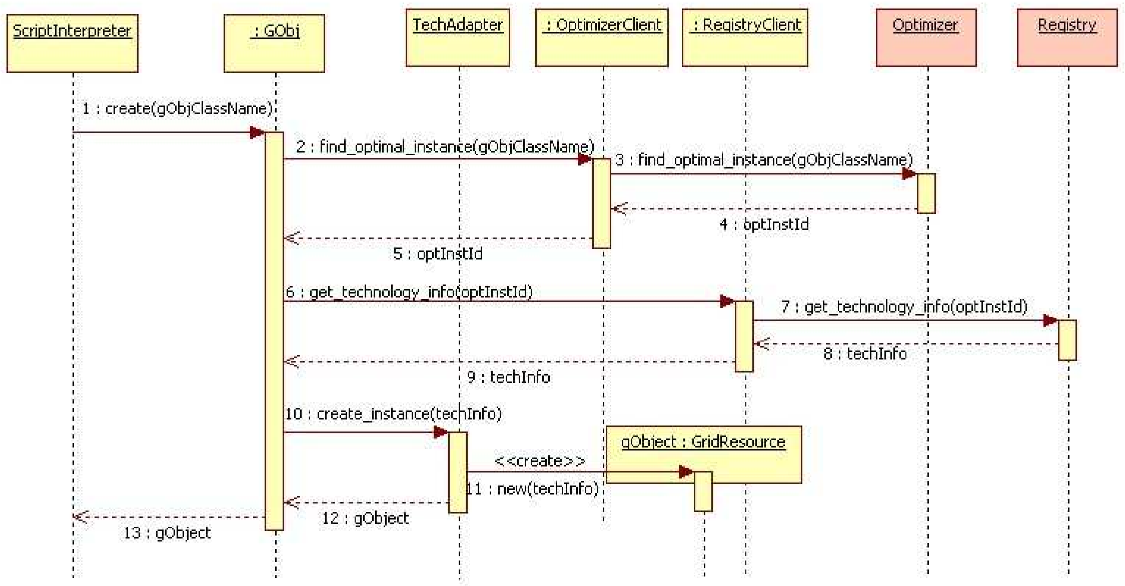


Figure 6.4. Sequence diagram 1: Creating a *Grid Object* of a given *Grid Object Class*.

6.6.2. Creating a *Grid Object* for a given *Grid Object Instance*

The process of creating a *Grid Object* for a given *Grid Object Instance* is similar to the one described in Section 6.6.1. In this case, a developer provides a unique identifier of the *Grid Object Instance*, therefore *Optimizer* is not used. The rest of the process is analogous to one described in Section 6.6.1. Fig. 6.6.2 shows the steps involved in this scenario.

Description of steps:

1. *create_instance* class method of *GObj* is used to request creation of a *Grid Object* for a *Grid Object Instance* which identifier is passed as an argument.
2. **Steps 2-9** are analogous to **steps 6-13** described in Section 6.6.1, and therefore are not explained here.

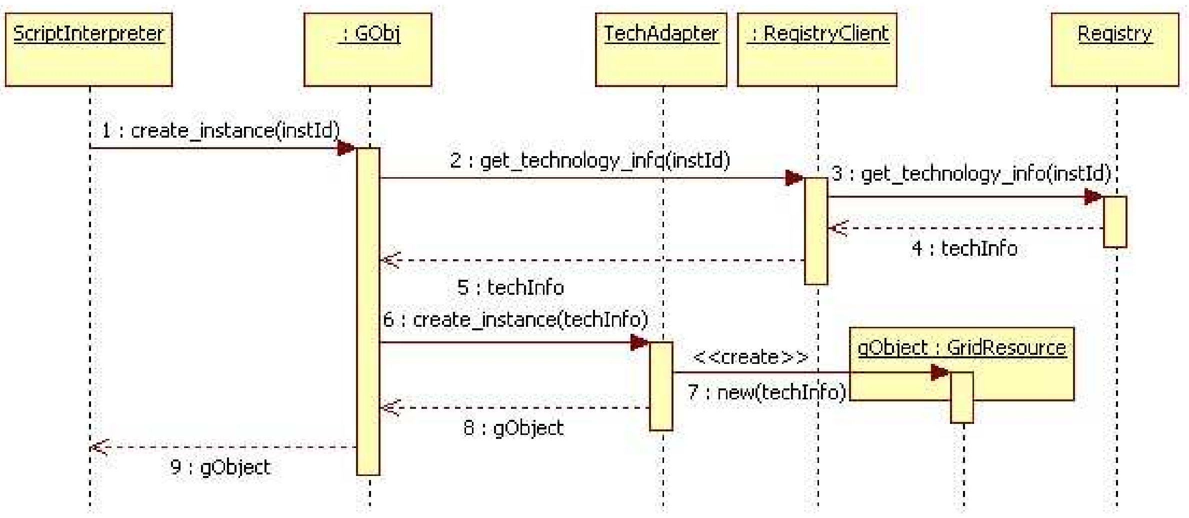


Figure 6.5. Sequence diagram 2: Creating a *Grid Object* for a given *Grid Object Instance*.

6.6.3. Invoking a synchronous and an asynchronous *Grid Operation*

Using *Grid Objects* in a source code of the experiment is identical to using ordinary Ruby objects, albeit the actual computation requested by calling a *Grid Operation* is performed on a remote machine. Developers can invoke operations in a synchronous or asynchronous mode. In the latter case, they have to remember that the nonblocking call returns a future variable and that they have to extract the actual result from it. Using asynchronous call is suitable for experiments that require to perform several pieces of computations that in an ideal case are independent (operate on disjunctive data and are executed on different machines). Fig. 6.6 presents synchronous invocation of a *Grid Operation* (steps 1-4) and an asynchronous one (steps 5-13).

Description of steps:

1. A *Grid Operation* is called on a *Grid Object* in an ordinary manner using RPC semantics.
2. The request is delegated by the *Grid Object* of an appropriate resource class to a *Grid Object Instance*. This is done in a protocol specific for the instance, data conversions and adding information required for invoking operation on the specific instance are added automatically.

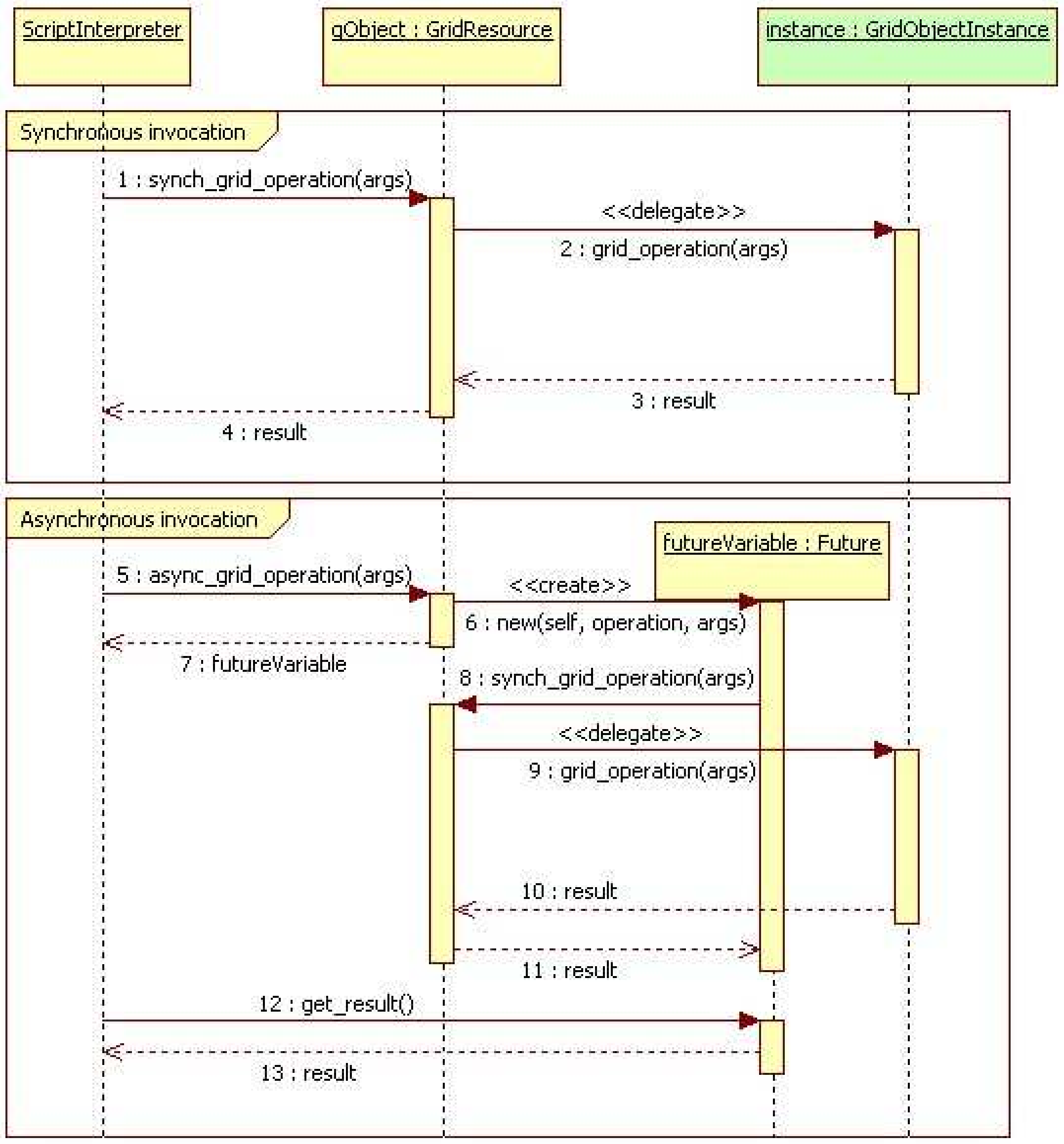


Figure 6.6. Sequence diagram 3: Invoking *Grid Operations* on a *Grid Object* in a synchronous and asynchronous manner.

3. The computation is performed by the *Grid Object Instance* on a remote machine and the result is returned in the instance specific protocol.
4. The result is converted to Ruby format and returned by the *Grid Object*.

5. A *Grid Operation* is called on a *Grid Object* in an ordinary manner using RPC semantics, the name of the operation is preceded with prefix *async_*. This informs the GOI library that this operation is invoked in an asynchronous mode.

6. A future variable is created. It receives a reference to the *Grid Object*, the name of the method and operation arguments as the parameters for the constructor.
7. The future variable is returned, but the actual result is not ready yet. The interpretation may continue.
8. The future variable call a synchronous *Grid Operation* on a *Grid Object* using the reference, operation name and arguments passed to the constructor.
9. The request is delegated to the *Grid Object Instance* (see step 2).
10. Result of the computation is returned to the *Grid Object* (see step 3).
11. Result is converted to the Ruby format and returned to future variable (compare to step 4).
12. The *get_result* blocking method is called on the future variable to extract the result (if this is called before the result is ready the interpreter waits for the synchronous call of the *Grid Operation* to complete).
13. The result is returned.

6.7. Interfaces provided by the Grid Operation Invoker

To preserve ease of use, the main functionality of the system, which is the ability to create *Grid Objects*, is delivered by a simple application programming interface. The high-level API is provided by a single class named *GObj*. It is the only class that the developer needs to use in a grid application and allows creating *Grid Objects* either by providing a name of a *Grid Object Class* or an identification of a *Grid Object Instance*.

Besides that, developers can use low-level API provided by the resource classes (*WsResource*, *MoccaResource*, *WtsResource*, *LcgResource*). They load a resource class and create a *Grid Object* directly by invoking resource constructor method and providing technology information. In case of MOCCA technology, resource class enables developer to manage components (create, destroy, connect ports etc.) in order to build an application of several component instances. The low-level API allows prototyping and testing *Grid Object Instances* that are not registered yet.

A detailed description of the Grid Operation Invoker API is given in Appendix B.

6.8. Interfaces required by the Grid Operation Invoker

Dependencies of the Grid Operation Invoker have been identified in Section 1.3.3. System requires an optimizer, which finds an optimal *Grid Object Instance*, and a registry, which stores technology information about instances. These needs can be satisfied by simple local implementations, however it is recommended to use external systems that employ better algorithms and use a data base in order to achieve better performance and scalability of the solution. The GOI system uses an *Optimizer-Client* and a *RegistryClient* that are responsible for interfacing an optimizer and a registry. Therefore our system does not require an optimizer or a registry to provide GOI specific interfaces, but only client classes for an optimizer and a registry. It is necessary that:

1. an optimizer client implements a method *find_optimal_instance* that accepts a *Grid Object Class* name and returns an identifier of the instance that can be used to query a registry;
2. a registry client implements a method *get_technology_info* that accepts an identifier returned by the optimizer client and return a Ruby Hash containing technology information about the *Grid Object Instance* pointed by the identifier.

Every client is dedicated to work with a specific optimizer or registry. The Grid Operation Invoker can be easily customized to use another clients that interface another optimizer and registry (see Appendix A.5).

Due to JRuby being an implementation technology, optimizers and registries might be implemented in Ruby, Java or be any remote software published with a technology supported by GOI (Web Service or MOCCA, job-oriented middleware technologies are not suitable for this purpose).

6.9. Summary

In this chapter we have presented the design of the Grid Operation Invoker. We have chosen to implement our system in the JRuby language. It is based on the broadly accepted Ruby scripting language and enables to use Java objects within the script. The GOI system has a modular architecture that allows customization and extendability. It queries an optimizer for an optimal *Grid Object Instance*, then retrieves technology information from a registry and loads an adapter dedicated for the instance. The adapter produces a *Grid Object*. Our solution enables to create *Grid Objects* either by providing a *Grid Object Class*, providing an identifier of an instance or using low-level API. *Grid Objects* can be used as an ordinary Ruby objects, albeit the computation is performed remotely. Developer can use stateless or stateful interaction mode and invoke operations synchronously or asynchronously. The system exposes its functionality by its main class, which is the *GObj*, and by the resource classes. The Grid Operation Invoker depends on an optimizer and a registry. It uses clients dedicated for these components, therefore can be customized to use various optimizers and registries.

Chapter 7

Validation of the Grid Operation Invoker

In this chapter we present the validation of the Grid Operation Invoker. First, we describe the main achievement of the thesis, which is providing uniform access to computational resources of the grid environment within a scripting language. Next, we depict provided functionality and system properties with regard to requirement defined during the analysis phase. Subsequently, we define testing approach, describe used testbed and list tests of the system. Finally, we discuss optimization issues and summarize this chapter.

7.1. Accessing *Grid Object Instances* published with diverse middleware technologies

Main objective of this thesis is to allow accessing the grid environment within a scripting language (see Section 1.2). In order to achieve this goal, we had to overcome the problem of using multiple middleware packages in a uniform manner (see Section 1.3.1). Our solution of the problem defined in Section 1.3 is based on the *Grid Object* abstraction defined in Section 4.1 and the idea of accessing the grid environment on multiple levels of abstraction in a uniform manner (see Section 4.2).

We have designed and implemented the *Grid Operation Invoker* system that supports the abstraction over the grid environment and enables to create and use *Grid Object Instances* using multiple middleware packages. It facilitates developing high-level applications, because it allows to access the grid environment as easy as in the code snippet presented in Fig. 4.2.

7.2. Provided functionality

During the analysis phase, we have identified use cases of the system (see Section 5.2). Grid Operation Invoker supports all these use cases, which can be summarized as follows:

- creating a *Grid Object* for a given *Grid Object Class*;
- creating a *Grid Object* for a given *Grid Object Instance* by providing its unique identifier;
- creating a *Grid Object* using low-level API;
- using *Grid Objects* in an ordinary manner;
- invoking stateless, stateful, synchronous and asynchronous *Grid Operations*;
- Manage components (create, destroy, connect and disconnect them).

The Grid Operation Invoker supports all these use cases. It enhances the JRuby scripting language with the ability of accessing heterogeneous computational resources in a coherent manner, thus facilitates developing high-level object-oriented grid applications. Extending a broadly accepted and powerfull scripting language allows fast development of concise, easy to understand and portable source code that may include application logic. The API provided by the system (see Appendix B) enables to work both on high-level of abstraction and to have full control if needed. Interfacing various middleware technologies during execution of the script is absolutely transparent from the enduser's point of view. At a runtime the system may select optimal computational resources, interface them in their specific protocols and make any necessary data conversions. The GOI library added asynchronous calls to all technologies that it supports. In addition, it allows to use job-oriented middleware packages in an object-oriented style.

7.3. Nonfunctional properties of Grid Operation Invoker

The GOI system satisfies the nonfunctional requirements defined in Section 5.3.2. It is based on the most accepted technologies such as Java and Ruby scripting language. It is able to interface *Grid Object Instances* published by Web Service, MOCCA, WTS and LCG middleware suites using standardized communication protocols. Ruby notation is very clear and concise. Development of high-level application is very easy (see example applications presented in Chapter 8). The GOI system can be easily extended by providing additional adapter classes (see Appendix C). This does not require any code modifications in the existing installation of the system.

Our solution is a light-weight and easy to install system. Its size is small (the size of the system depends on the client-side libraries required for middleware technologies). What is more the system does not start any system services, nor require any incoming ports to be opened. The system can be used on any modern client machine. Creating *Grid Objects* and handling invocations of *Grid Operations* are neither computation, nor memory intensive. The Grid Operation Invoker is platform independent client-side library. It can be downloaded as a JRuby library or as single package containing a JRuby interpreter packaged with the GOI library (for more detailed information on GOI distributions please refer to Section A.3).

The GOI system is able to work as a standalone system. Besides that, it has well defined provided interface and dependencies. It can be customized to cooperate with a variety of external optimizers and registries. Presented solution enables to solve problems from a wide range of domains (see data mining and virology applications presented in Chapter 8). The Grid Operation Invoker has been applied as a core of the runtime system of the ViroLab Virtual Laboratory [43], [44].

7.4. Tests

7.4.1. Testing approach

Tests of the Grid Operation Invoker systems involves testing support for every single middleware package, testing programming properties added by the GOI system (see Table 2.3) and implementing experiments employing *Grid Object Instances* published with diverse middleware packages. For each middleware suites we have checked if the following are possible:

1. creating a *Grid Object* for a *Grid Object Instance* exposed by this technology using every possible way: specifying the *Grid Object Class*, providing an identifier of the specific *Grid Object Instance* using this technology and using low-level API of resource class;
2. invoking *Grid Operations* on a *Grid Object* likewise ordinary Ruby objects;
3. reporting errors specific for this technology.

Subsequently, we have tested asynchronous invocations of *Grid Operations* and using job-oriented technologies in a object-oriented manner. Finally, in order to test interoperability among various middleware packages we have implemented experiments solving real life problems that use more than one middleware suite (see Chapter 8).

7.4.2. Description of the testbed

Client machine

The Grid Operation Invoker is a client-side software. It was installed and tested on the following machines:

- IBM Thinkpad T42 laptop (Intel(R) Pentium(R) M Processor @ 1.70GHz, 512 MB RAM) under Kubuntu 7.10 and Microsoft Windows XP Professional operating systems;
- desktop personal computer (AMD Athlon 3000+ processor, 512 MB RAM) under Kubuntu 7.10 and Microsoft Windows XP Professional operating systems;
- server machine - *virolab.cyfronet.pl* (2 dual core Intel(R) Xeon(R) 5150 processors @ 2.66GHz, 4 GB RAM) under Kubuntu 6.06.

Server machines

In order to test all technologies that are supported by the Grid Operation Invoker system the following machines were used:

- server machine - *virolab.cyfronet.pl* (2 dual core Intel(R) Xeon(R) 5150 processors @ 2.66GHz, 4 GB RAM) under Kubuntu 6.06 was used to test Web Services (Tomcat 5.5 container) and MOCCA framework (H2O 2.1 kernel);
- EGEE testbed accessed via a machine with EDG User Interface installed to test LCG middleware package;
- server (*virolab.med.kuleuven.be*) with a Witty Service running;
- server (*ws.virolab.org*) with a native Ruby Web Service container.

Grid Operation Invoker configuration

While testing, the system was configured to use:

- the *remote_registry_client* registry client that delegates queries to the remote Grid Resource Registry (Web Service endpoint: *http://virolab.cyfronet.pl:8080/grr-0.2.3/services/InvocationDataProvider*);
- the *scheduler_client* optimizer client that delegates queries to a Grid Application Optimizer (local Java application).

7.4.3. Supported middleware technologies

We consider a technology supported if a *Grid Object* for an instance using this middleware suite can be created and used in a uniform manner. To prove that a middleware packages is supported we provide a code snippet that creates a *Grid Object* and invoke a *Grid Operation*. All these snippets use high-level API, which depends on lower-level API, therefore all three method for producing *Grid Objects* (see Section 5.2.1) are tested. Below we list supported technologies and code snippets testing them.

- **Web Service**[1]. Testing Web Service middleware is presented by lines 2-4 in Fig. 7.1. Line: 2.) Creating a *Grid Object* 3.) Calling a *Grid Operation* that returns given message 4.) Printing the result.
- **WSRF** Testing WSRF technology is illustrated by lines 7-9 in Fig. 7.1. Line: 7.) Creating a *Grid Object* 8.) Calling a *Grid Operation* that adds *5* 9.) Printing the current value.
- **MOCCA** Testing MOCCA middleware is illustrated by lines 12-14 in Fig. 7.1. Line: 12.) Creating a *Grid Object* 13.) Calling a *Grid Operation* that returns given message 14.) Printing the result.
- **Witty Services**. Testing WTS middleware is depicted by lines 17-19 in Fig. 7.1. Line: 17.) Creating a *Grid Object* 18.) Calling a *Grid Operation* that returns the alignment for a given nucleotide sequence with respect to the given region 19.) Printing the result.
- **LCG**[2]. Testing LCG middleware is shown by lines 22-24 in Fig. 7.1. 22.) Creating a *Grid Object* 23.) Calling a *Grid Operation* that returns the output of *ping*

[1] Presented code tests a service that uses a RPC binding style and is not described with a WSDL, however services with a WSDL descriptions and those using DOCUMENT binding style are supported as well.

[2] Please note that scripts accessing the LCG suite must be executed on a machine with installed EDG User Interface and a valid proxy certificate must exist.

command sending 5 ECHO requests to a host given as parameter 24.) Printing the result.

```
1  #Web Service
2  echo = GObj.create('cyfronet.gridspace.test.EchoService')
3  reflected = echo.echo('Hear me roar!')
4  puts 'Reflected message: ' + reflected
5
6  #WSRF
7  math = GObj.create('cyfronet.gridspace.test.Math')
8  math.add(5)
9  puts math.getValueRP(GetValueRP.new())
10
11  #MOCCA
12  hello = GObj.create(cyfronet.gridspace.test.Hello)
13  msg = hello.echo(Works fine.)
14  puts Hello message:  + msg
15
16 #WTS
17 alignTool = GObj.create('regadb.RegaAlignment')
18 alignment = alignTool.align(nt_seq, 'PRO')
19 puts alignment
20
21 #LCG
22 ping = GObj.create('cyfronet.gridspace.test.LcgPing')
23 result = ping.ping('virolab.cyfronet.pl')
24 puts result
```

Figure 7.1. Testing GOI support for various middleware.

In order to make these code snippets runnable application the following line is required at the beginning of the script:

```
require 'cyfronet/gridspace/goi/core/g_obj'
```

Moreover in the WTS test a *nt_seq* variable must be defined.

As presented in code snippets (see Fig. 7.1) these four technologies are supported. *Grid Objects* are created in a uniform manner and used within the script as ordinary Ruby objects.

7.4.4. Testing added programming properties

The Grid Operation Invoker has added two programming properties:

1. asynchronous invocation of *Grid Operations* for all supported technologies;
2. support for object-oriented programming paradigms for job-oriented middleware suites (WTS and LCG).

First property has been tested by implementing an asynchronous version of the Weka experiments (see Section 8). Results of the asynchronous version have been compared to those produced by a synchronous application. The script execution has been inspected to find out if the execution of the asynchronous script is suspended only in case if the requested result is not ready yet. The speedup of application was noticeable.

The second one does not require any testing. *Grid Object Instances* published by the WTS and LCG job-oriented middleware packages are used in an object-oriented manner (see code snippets testing support for WTS and LCG middleware suites in Section 7.4.3)

7.5. Optimization issues

Choosing the optimal instance is done by an optimizer. GOI can use various optimizers. Developers can choose instances directly by providing theirs identifiers. Developers are aware of programming properties of instances, therefore can implement more efficient applications.

7.6. Summary

In this chapter we have described the validation of the implemented system. The Grid Operation Invoker supports the Grid Object abstraction defined in Section 4.1 and realizes the concept of accessing the grid environment within a JRuby scripting language (see Section 4.2). It supports all use cases defined in Section 5.2 and satisfies both the functional and nonfunctional requirements (see Section 5.3). We have defined criterion for a technology to be considered as supported: it must be possible to create *Grid Objects* representing instances published by this technology using uniform interface and these *Grid Objects* must be identical in usage as ordinary objects. Subsequently, we proved that the Web Services, WSRF, MOCCA, WTS and LCG packages are supported by the GOI system.

Chapter 8

Grid applications using the Grid Operation Invoker

In this chapter we present two applications: the Weka data mining and the HIV genotype to drug ranking system. We enumerate what computational resources, which are published with diverse middleware, are used and prove that these resources are accessed in a uniform manner using the GOI library. Finally, we provide conclusions stemming from development of these grid applications.

8.1. Weka data mining application

Weka [45] is a set of tools for data mining tasks that can be applied to scientific problems in a variety of domains. It allows data pre-processing, classification, regression, clustering, association rules, and visualization. *Weka* is a broadly accepted software that is used both for educational purposes and research.

We have implemented a sample data mining applications using the Grid Operation Invoker to access remotely the functionality of the *Weka* library (see Fig. 8.1). This application is based on an the example from [45]. It illustrates a typical data mining use case and solves a so called *The weather problem.* A classifier employing *One-Rule* (*1-R*) algorithm is used to predict whether certain conditions are suitable for playing an unspecified game. The application involves the following steps:

- retrieving data in a *Weka* specific format,
- splitting the data into a training and testing data sets,
- training a *One-Rule* Classifier,
- classifying the testing data,
- evaluating the prediction quality.

```
require 'cyfronet/gridspace/goi/core/g_obj'

retriever =  GObj.create('cyfronet.gridspace.gem.WekaGem')

A = retriever.loadDataFromDB(DATABASE, QUERY, USER, PASSWORD)
B = retriever.splitData(A, 20)
trainA = B.trainingData
testA = B.testingData
classifier = GObj.create('cyfronet.gem.weka.OneRuleClassifier')
attributeName = 'play'
classifier.train(trainA, attributeName)
prediction = classifier.classify(testA)
logger.info('Predicted data:' + prediction.to_s)
quality = retriever.compare(testA, prediction, attributeName)
puts ('Prediction quality:' + quality.to_s)
```

Figure 8.1. Weka data mining application.

The *Weka data mining* application uses the following *Grid Object Instances*:

- The instance of *cyfronet.gridspace.gem.weka.WekaGem Grid Object Class* that retrieves the data in the ARFF format [46], splits the data and evaluates prediction quality. It is published using the Web Service middleware;
- The Instance of *cyfronet.gridspace.gem.weka.OneRuleClassifier* class that classifies the data (it is necessary to train the classifier prior to classifying data). It is published using the MOCCA middleware.

8.2. HIV genotype to drug ranking

This application has been developed within the *ViroLab* project [1] and it solves a real life problem in the virology fields. It enables virologists to find the *HIV* susceptibility for drugs based on the genotype of the virus. This application can be used by a virologist to conduct research as well as to advise medical doctors how to enhance and personalize a treatment for a *HIV* infected patient.

The process of interpreting HIV genotype to drug resistance consists of the following steps:

- aligning a *HIV* nucleotide sequence with respect to a reference strain,
- detecting subtype of the virus,
- finding mutations in an indicated region,
- using *ViroLab Drug Ranking System* to find out the drug resistance of the virus.

Fig. 8.2 presents the source code of the *HIV genotype to drug ranking* application.

This application uses the following *Grid Object Instances*:

- The instance of *regadb.RegaAlignment Grid Object Class* that aligns the nucleotide sequence with a reference strain. It is published using the WTS middleware.
- The instance of *regadb.RegaHivSubtype* class that determines the subtype of the virus. It is published using the WTS middleware.
- The instance of *org.virolab.DrugRankingSystem* class that provides virus drug resistance data. It is published using the Web Service middleware.

```
require 'cyfronet/gridspace/goi/core/g_obj'

region = "rt"

#this data in not complete!
ntSeq = (">55106  Tue Jun 05 19:56:52 PDT 2001. 1269 bases.\n
CCTCAGATCACTCTTTGGCAACGACCCMTCGTCACAATAAAGGTAGGGGG
......
TTATGACCCATCAAAAGAC")

regaDBMutationsTool = GObj.create('regadb.RegaAlignment')
mutations = regaDBMutationsTool.align(ntSeq, region.upcase)

regaDBSubtypingTool = GObj.create('regadb.RegaHivSubtype')
subtype = regaDBSubtypingTool.subtype(ntSeq)

mut = mutations.split(',').last.chop
drs = GObj.create('org.virolab.DrugRankingSystem')
puts drs.drs('retrogram', region, 100, mut)
```

Figure 8.2. HIV genotype to drug ranking.

The *HIV genotype to drug ranking* application illustrates an advantage of the scripting approach to building grid applications. One can see that *mutations* returned by the *RegaDB Mutations Too*l can not be directly passed as an input to the *Drug Ranking System.* A very simple data conversion is required. It is not a computationally intensive process therefor is done locally in a one line of code. A combination of using remote computational resources with local processing and application logic defined in a scripting programming language is in our opinion the most flexible and suitable approach for building grid applications.

8.3. Summary

We have implemented applications that deal with real life problems from two scientific domains: data mining and virology. We have demonstrated how easy building grid applications utilizing the Grid Operation Invoker library is. Each application access in a uniform manner *Grid Object Instances* published with different technology, thus interoperability among grid middleware is illustrated. Virology and data mining applications proved that the Ruby language combined with the GOI library constitutes a general purpose solution that allows building grid applications for a wide range of scientific domains.

Chapter 9

Summary and future work

This chapter includes a summary of the work done in scope of providing uniform interface to grid resources and plan for possible evolution of the Grid Operation Invoker system in future.

9.1. Thesis summary

The objective of this work was to provide a uniform interface for accessing grid resources within a scripting language. We have conducted the research in this scope and developed a prototype implementation of the system. The work has been carried with respect to the best practices in research and software engineering. The latest but reliable technologies have been used.

We have analyzed the requirements for the system (see Chapter 5) and on this basis we have designed the Grid Operation Invoker system and implemented (see Chapter 6) an extendable Ruby library that enables to use computational resources published with a variety of middleware technologies. The prototype implementation satisfies both functional and nonfunctional requirements and supports Web Service, MOCCA, Witty Services and EGEE jobs (see Chapter 7). Finally, we have developed grid applications using the GOI library.

The objectives of the thesis (see Section 1.4) have been achieved. The Grid Operation Invoker system is operational, it is integrated with a remote Grid Resource Registry [47] and a Grid Application Optimizer [48], and is used as a core of the runtime system of the ViroLab Virtual Laboratory [43]. The GOI has been applied to real life problems in fields of data mining and virology (see Chapter 8). This proved that the GOI library facilitates implementation of object-oriented grid applications and illustrated interoperability among heterogeneous middleware. What is more, grid applications expressed in a Ruby scripting language enhanced with the GOI library are concise and easy to understand. Grid applications combine local processing with remote computation and can be easily executed on every modern personal computer. The scripting approach appeared to be suitable for development of high-level grid applications.

9.2. Future work

The work on the Grid Operation Invoker will be carried on. First of all, we will work on enhancing adapters and add support for more middleware technologies. WSRF adapter will be enhanced with ability to dynamically generate stubs for stateful services. Next, we will work on adapters for AHE and Unicore in order to provide access to computational resources of large European grid infrastructures. At the same time we will integrate more *Grid Object Instances* and implement more complex scripts.

Furthermore, we will consider providing introspection mechanisms that would allow obtaining information usefull for developers. This data would include for instance a list of *Grid Operations* and their semantic meanings. Introspection mechanism is especially important while working with dynamic scripting languages, like Ruby, because these languages are untyped. Once this mechanism is enabled, developers will be able to develop and execute experiments interactively, thus will fully exploit advantages of dynamic scripting language.

Another possible way of GOI evolution is integration with security mechanism. The most promising candidate is the Shibboleth system [49]. It is a federated Single-SignOn and attribute exchange framework that enables to authenticate to a local identity provider and access resources of many providers. The Grid Operation Invoker would include a required security token in invocations of *Grid Operations* transparently.

The last idea on enhancing the GOI system is to integrate it with the monitoring system. While executing a script the GOI library would log information regarding scientists executing experiments, used *Grid Object Instances*, operation inputs and produced results. Logs would be delivered to a monitoring system by remote appenders. Gathered information would allow provenance as well as provide significant data for an optimizer.

Appendix A

Installation guide

This chapter covers in detail installation of the Grid Operation Invoker system and all of its prerequisites on both the Linux and Windows platforms. Next, it explains how to customize the system to use various registries and optimizers.

A.1. Prerequisites

The Grid Operation Invoker can be installed and run on any modern computer using any operating system, provided that the machine has network connectivity. The Grid Operation Invoker is implemented as a JRuby library, therefore an interpreter (standard distribution or the one included in the **JRuby plus GOI**distribution) must be installed and configured. JRuby interpeter can be downloaded free of charge from [50]. Since JRuby is a Java implementation of Ruby interpreter it requires a **Java Runtime Environment**, which can be downloaded from [51]. The Grid Operation Invoker software was implemented and tested using the JRuby version 1.0 and JRE 1.6.0, thus these releases are strongly recommended. Make sure that JAVA_HOME and JRUBY_HOME environment variables are set up properly.

A.2. Dependencies

GOI uses some Java libraries that provide client side access to Grid Object Instances in specific middleware technologies, such as MOCCA or WTS. Theses libraries introduce some indirect dependencies to other jars. MOCCA relies on the *mocca_light.jar* library, which depends on the *H2O* jars and configuration files. Witty Services requires a *wts-client-java-0.9.jar* to be used. Both, the Grid Operation Invoker direct and indirect, jar dependencies are described in the

Maven [52] Project Object Model files, which can be found at [53].

Besides client side libraries, the GOI system requires a grid certificate, EDG User Interface installation with its executables in the $PATH environmental variable to enable running jobs on the EGEE testbed. Before experiment is started user needs to create his proxy certificate (by executing *grid-proxy-init* command and provide the passphrase if requested). For more instructions on acquiring a grid certificate and using LCG please refer to [54].

A.3. Grid Operation Invoker distributions

Grid Operation Invoker is an extension of the JRuby scripting language. There are two types of distributions:

1. **JRuby plus GOI**, which includes interpreter and the GOI library with all necessary Java libraries
2. **GOI**, which contains the GOI libraries only

Both distributions satisfy all dependencies except the EDG user interface, which needs to be installed separately.

A.4. Installation

There are two different distributions so choose one of the them and follow the instructions for it. First of all, make sure that all prerequisites are satisfied. Next, choose the distribution type. If a JRuby interpreter has not been already installed choose the interpreter enhanced with the GOI libraries(**JRuby plus GOI**). In case JRuby is already installed choose the **GOI** distribution. To install the Grid Operation Invoker on a machine follow instructions for a suitable distribution.
Installation procedures were successfully tested on Kubunt 7.04 Linux, Kubuntu Linux 7.10 and Windows XP Professional operating systems with JRE 1.6.0 installed.

A.4.1. Installing JRuby plus GOI

1. download distribution from [53] - if *wget* is available issue the following command in a console:
   - *wget http://virolab.cyfronet.pl/ tomek/msc/software/jruby-1.0-plus-goi.tgz* on Linux
   - *wget http://virolab.cyfronet.pl/ tomek/msc/software/jruby-1.0-plus-goi.zip* on Windows

   or use your favorite web browser to download the appropriate file
2. move downloaded file to the location where you wish to install the system
3. extract the archive
   - execute *tar xfv jruby-1.0-plus-goi.tgz* on Linux
   - open a context menu for the zip archive and choose to extract it
4. set *JRUBY_HOME* variable

   - for Linux append the following line to the *.bash_profile* in the home directory: *export JRUBY_HOME=path-to-the-jruby-plus-goi*
   - for Windows open *Start Menu -> Settings -> Control Panel*, double click *System* and choose *Advanced* tab, click *Environment Variables*, create a new one called *JRUBY_HOME* and assign a value that point to the installation directory
5. restart the system to make sure new environment variable is used
6. delete the downloaded archive

A.4.2. Installing GOI

1. download goi.tgz from [53] - if *wget* is available issue the following command in a console
   - *wget http://virolab.cyfronet.pl/ tomek/msc/software/goi.tgz* on Linux
   - *wget http://virolab.cyfronet.pl/ tomek/msc/software/goi.zip* on Windows

   or use your favorite web browser
2. move downloaded file to *JRUBY_HOME*
3. extract archive
   - by typing in the console *tar xfv goi.tgz* on Linux
   - by opening a context menu for the zip archive and choosing to extract it

A.4.3. Testing GOI

Change directory to the GOI installation and execute the following command in the console *jruby test/test_suite_all.rb*. If this test is passed then you have successfully configured the Grid Operation Invoker system on your machine.

A.5. Customizing the Grid Operation Invoker

The Grid Operation Invoker has a modular architecture. The core part of the system delegates the queries to registry and optimizer clients, thus by replacing them, it is able to cooperate with a diversity of registries and optimizers. The system is configured by the constants that are defined in the *constants.rb* file, which is placed in the *cyfronet/gridspace/goi/core* directory. Table A.1 provides a full listing of currently used constants.

In addition, more constants can be defined and used in the GOI source code. Due to configuration file being a Ruby source, constants' values can be expressions evaluated at run-time.

Constant name	Explanation
OPT_CLIENT_CLASS	A name of the class that is responsible for delegating queries to an optimizer and returning results. This class will be required from *$JRUBY_HOME/lib/ruby/1.8/cyfronet/gridspace/goi/core/* directory.
REG_CLIENT_CLASS	A name of the class that is responsible for delegating queries to a registry and returning results. File containing this class will be searched in the same location as optimizer client class.
GS_GRR_BASE_URL	An endpoint of the Grid Resource Registry.
GS_GRAPPO_URL	An endpoint of the Grid Application Optimizer.

Table A.1. Constants customizing Grid Operation Invoker

Appendix B

Grid Operation Invoker API

This chapter briefly explains the principle of operation and introduces the API that Grid Operation Invoker exposes to the developer of the experiment. It explains different levels of abstraction and provides suggestions which level should be used.

B.1. Principle of operation of the Grid Operation Invoker

GOI provides uniform interface to invoke operations on Grid Objects Instances, that can be published using various middleware technologies, such as Web Services, MOCCA components, jobs etc. To fulfill its responsibilities GOI performs the following activities while creating Grid Object representative:

- querying Optimizer for id of optimal Grid Object Instance of the class requested in the script (experiment)
- querying Registry for technical information describing selected instance
- loading appropriate technology adapter, which creates Grid Object representative

Once created, Grid Object can be used just like any other JRuby object. The burden associated with interfacing specific middleware is covered from the user. GOI makes invoking remote operation identical to calling a method on a local object.

It is possible for developer to bypass listed steps if she/he wishes to use low level API provided by adapters. There are three possible ways of GOI usage:

1. create Grid Object of a given class
2. create Grid Object for a given Grid Object Instance
3. create Grid Object using low level API by providing all necessary technical information

Whichever manner is used, the returned object is always an object representative of a piece of software deployed on a remote or local computational resource.

B.2. GObj API

First of all, developer must require necessary Ruby files. *GObj* class is the most essential and in most cases, excluding the third scenario, it is the only class that needs to be loaded explicitly in the experiment source code. To do so, developer must include the following code:

```
require 'cyfronet/gridspace/goi/core/g_obj'
```

After that, creating Grid Object Instance Representatives using *GObj* class methods: *create* and *create_instance* is possible. The former method take the name of the Grid Object Class as an argument. Such invocation performs all three steps mentioned in section describing GOI principle of operation. For instance, to create a representative for a Grid Object Instance of class named cyfronet.gridspace.gem.EchoService the following code would be used:

```
echo1 = GObj.create(cyfronet.gridspace.gem.EchoService)
```

Now, let us do the same using the latter method, create_instance, which takes id the Grid Object Instance.

```
echo2 = GObj.create_instance(5)
```

In this case, querying Optimizer for an optimal instance is omitted, but developer must be sure that the id of desired Grid Object Instance , which is stored in the Registry, equals 5. Otherwise it is possible that later in experiment there will be an attempt to invoke operation which is not provided by the Grid Object Instance and an error will be raised.

B.3. Low level adapter API

Let us create the same object using the last option, the low level adapter API, which requires much more effort. To begin, we must load the appropriate resource class, in this tutorial we will create a representative for Web Service:

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

Next, the technical information is needed (see D). Below a Hash containing the full information about Grid Object Instance is defined.

```
techInfo = {'instId' => 5, 'name' => 'instance1',
            'endpoint' => 'http://virolab.cyfronet.pl:18080/',
            'type' => 'WS','wsType' => 'RPC',
            'method#0' => 'echo','in#0#0' => 'echoString',
            'out#0#0' => 'echoReturn',
            'namespace' => 'http://virolab.cyfronet.pl/echo',
            'codebase' => 'url'}
```

Please remember, that technology information is specific for every middleware. Finally, let us create the resource representing the EchoService:

```
echo3 = WsResource.new(techInfo)
```

B.4. Choosing the appropriate API

As already proved in this appendix, using different APIs involves various levels of knowledge and understanding of GOI. Although all methods for creating Grid Object Instance representatives provide the same functionality and produce the same result, they provide different non-functional capabilities. For instance, the *create* method is the most convenient to use and the most universal, while the *create_instance* can be used to ensure that a specific instance will be used, because of accounting issues, reliability or numerical quality of the software installed on a concrete node, etc. In the end, the low level API can be used for testing new GEMs, before they will be registered in GRR, as well as using external Grid Objects Instances.

B.5. Using Grid Object representatives

Variables *echo1*, *echo12* and *echo3* are representatives of the same Grid Object Instance. They can be used identically, albeit they were produced using distinguishable methods. Moreover, invoking operation on a representative is analogical to calling method on a ordinary JRuby object. Code snippet below compares usage of representatives and ordinary JRuby object.

```
ordinary = String.new('I am local object')
l = ordinary.length
puts l

msg1 = echo1.echo('I am easy to use!')
puts msg1
msg2 = echo2.echo('I am easy to use too!')
puts msg2
msg3 =echo3.echo('So am I!')
puts msg3
```

Appendix C

Implementation of technology adapters

This manual provides a step-by-step guide how to extend run-time capabilities of the Grid Operation Invoker by adding support for new middleware technologies. It explains in detail how to implement adapter, resource and wrapper classes.

C.1. Adding support for new technologies

Grid Operation Invoker supports leading Grid middleware technologies. Currently, its users can employ Web Services, MOCCA Grid components, EGEE jobs (LCG) and WTS in experiments. Moreover, design and implementation of the system make enriching the system with support for new technologies absolutely undemanding. It requires only two things to be done:

1. implementing adapter and resource classes for new technology and placing them in appropriate directory
2. preparing the registry to store technical information describing instances published in new technology

All classes of Grid Operation Invoker can be implemented in JRuby, therefore they can contain pure Ruby code, as well as include and use Java objects. If any Java classes, other than those provided by Java Runtime Environment, are used, jar files containing theses classes should be placed in *$JRUBY_HOME/lib/* directory.
For more information on Ruby language please refer to [40, 42].

C.2. Extending Grid Operation Invoker

Grid Operation Invoker support for various middleware technologies is based on adapters concept. The system can be extended by adding adapter and resource classes for new technology. Adapter is a factory that is capable to produce representatives for all Grid Object Instances published in one concrete middleware technology. Such representative should be an object of a resource class, that extends GridResource class that can be found in $JRUBY_HOME/lib/ruby/1.8/cyfronet/gridspace/goi/adapters/ directory. Resource object is a proxy for Grid Object Instance that is able to interact with instance in its specific protocol, however, from the experiment developer's point of view, resource objects act as any other JRuby object. Adapter class must expose one class method, create_instance, which takes Hash object containing technology data and returns Grid Object Instance representative of *TechnologyResource* class. Resource class must implement only one method, initialize taking Hash with technology information. This method semantics is very similar to Java constructor, it is called after Ruby allocates memory for an objects and all arguments passed to *new* are passed to *initialize* method that sets up object state. The first thing that needs to be done in the body of this method is calling the *initialize* method of super class (*GridResource*) with he following code:

```
def initialize(techInfo)
 super(techInfo)
 # method body
end
```

Next, *TechnologyResource* object must be made capable to handle invocations of operations on Grid Object Instance. Developer is not limited in the way how to achieve it, albeit it is strongly recommended to implement the *method_missing operation*, which takes responsibility for delegating operation execution to Grid Object Instance and returning result.
Code snippets below present the simplest adapter and resource classes that support Web Service technology and explains what activities are performed.

```
require 'cyfronet/gridspace/goi/adapters/ws_resource.rb'
require 'java'
if !defined? JLogger
 include_class('org.apache.log4j.Logger'){|package,name| "J#{name}"}
end
class WsAdapter
 @@logger = JLogger.getLogger('goi.adapter.ws')
 def WsAdapter.create_instance(wsTechInfo)
   @@logger.debug('Using service ' + wsTechInfo['endpoint'])
   gridObjectInstance = WsResource.new(wsTechInfo)
   return gridObjectInstance
 end
end
```

Lets inspect what is done in this adapter:

1. *WsResource* class is required

2. Java support is enabled by requiring 'java'
3. log4j class is included if it not already defined
4. adapter class is defined
   - class variable, *@@logger* is defined that is used to print debug information and in future will be used to integrate GOI with monitoring infrastructure
   - representative for Grid Object Instance is created and returned

```
require 'cyfronet/gridspace/goi/adapters/grid_resource'
require 'java'
if !defined? JLogger
 include_class('org.apache.log4j.Logger'){|package,name| "J#{name}"}
end
class WsResource < GridResource
 attr_reader :soap
 def initialize(techInfo)
   super(techInfo)
   # ...
 end

 def method_missing(methodSymbol, *args)
   # ...
 end
end
```

The code above:

1. requires GridResource class, enables Java usage and include log4j class
2. *WsResource* class is defined
   - it extends *GridResource*
   - defines local variable *soap* and create getter method for it
   - defines *initialize* method (body of this method is not included)
   - defines *method_missing* that delegates operation invocations to Grid Object Instance and returns result (body of this method is not included)

Source code of *WsAdapter* and *WsResource* classes can be found in *ws_adapter.rb* and *ws_resource.rb* files placed in the *$JRUBY_HOME/lib/ruby/1.8/cyfronet/gridspace/goi/adapters* directory of the distribution.

Since adapter class is required during run-time, adapters' class names must obey certain naming convention. Due to the fact, that Ruby requires files, which names does not necessarily correspond to the name of class that is inside, naming convention is also imposed on file names. The naming pattern is as follows:

1. Adapter class name is a concatenation of two words. Technology name, starting with capital letter with all following letters in lower case, and 'Adapter'. For instance, adapter for Web Services is named *WsAdapter* (it could be named *WebserviceAdapter* as well, but *WebServiceAdapter* is not a valid name), adapter for MOCCA is named *MoccaAdapter*.

2. File name should reflect the name of the adapter class it contains according to the Ruby convention, that is to use only lower case letters and underscore characters (uppercase letter in class name must be replaced with underscore and lower case letter). Files must have 'rb' extension. File names for adapters mentioned above are *ws_adapter.rb* and *mocca_adapter.rb*.

It is not obligatory to name *TechnologyResource* classes in accordance to the mentioned naming convention, although it is profitable to name them analogically to adapter classes. For instance classes of representatives for Web Services is named *WsResource* and for MOCCA components is named *MoccaResource*, files containing these classes are *ws_resource.rb* and *mocca_resource.rb*.

Files containing adapter and resource classes should be placed in *$JRUBY_HOME/lib/ruby/1.8/cyfronet/gridspace/goi/adapters* directory. If any additional JRuby classes are used they should be placed in *$JRUBY_HOME/lib/ruby/1.8/cyfronet/gridspace/goi/utils.* These files should be required within adapter and resource classes like this:

```
require 'cyfronet/gridspace/goi/utils/additional_class_name'
```

C.2.1. Wrappers

Witty Services and LCG (for EGEE testbed) adapters are based on the concept of the **Local Gem**. It is impossible to map input parameters passed in the experiment to files that are required for jobs automatically, and vice versa, to map output files to results returned in the script by executed Grid Operation. The situation with WTS service is analogical to jobs. Therefore technical information for these technologies contains a JRuby class that is responsible for mapping input parameters and outputs. Developers who wish to register a new Grid Object Implementation using WTS or job technology is responsible for providing wrapper class. This section of manual is intended for them.

Wrapper class is required to provide two methods for every Grid Operation of Grid Object, named gridoperationname_submit and gridoperationname_get_output. The former is responsible for definition of WTS service that will be used or executable that will be run on EGEE testbed, as well as inputs mapping. It takes the input parameters that are passed to the representative of Grid Object Instance and uses *WtsSpec* or *JobSpec* class. The latter, defines what exactly is returned as the result of the Grid Operation invocation.

WTS wrapper

The submit method implementation must use WtsSpec class, which provides the following API:

1. *new* method enables the developer of the wrapper to:
   - Select the WTS service that will be used (first argument).
   - Enter user name and password (defaults are username=public, password=public)

- Choose the factory that will produce the WTS service endpoint (default is http://virolab.med.kuleuven.be/wts/services/)

   Only the first argument for the *new* method is compulsory.

2. setter methods for *fileoutputs*, *fileinputs*, *byteoutputs*, *byteinputs* fields that enable developer to define if communication with the service is based either on files or byte streams. It is obligatory to define both the inputs and the outputs of the service representative.

The get_output method takes the output of the WTS service and maps it to the Grid Operation result returned inside the JRuby script. To illustrate how to implement a wrapper let us focus on the RegaDB Alignment tool, which provides the align operation. Below there is a wrapper class that is stored in the registry.

```
class RegaAlignment
 def align_submit(ntSeq, protein)
   wtsSpec = WtsSpec.new('regadb-align')
   wtsSpec.byteinputs={'nt_sequences' => ntSeq, 'region' => protein}
   wtsSpec.byteoutputs=['aa_mutations']
   return wtsSpec
 end

 def align_get_output(output)
   return output['aa_mutations']
 end
end
```

In order to retrieve descriptions of RegaDB services an object of *WtsMetaClient* class should be used. A list of services can be obtained by calling *listServices* method. Next, more information about the service can be fetched using *getServiceDescription* method. The JRuby code below lists WTS services and collects information on the regadb-align service.

```
 require 'java'
 include_class('net.sf.wts.client.meta.WtsMetaClient')

 meta_client = WtsMetaClient.new(
"http://virolab.med.kuleuven.be/wts/services/"
)
 services = meta_client.listServices()
 #puts services
 jString = JString.new(
meta_client.getServiceDescription('regadb-align')
)
 puts jString
```

Wrapper for a job

Implementation of the *submit* method must utilize the *JobSpec* class that enables developer to provide a description of the job. Later on, this description will be used to generate a JDL file for the job. *JobSpec* provides:

- Setter methods for *executable*, *stdoutput*, *stderr* and *arguments* fields, that enables developer to define the executable that will be run on the EGEE testbed, standard output and error and arguments for the executable.
- *add_to_input_sandbox(input)* method that adds one given input to the input sandbox of the job.
- *add_to_output_sandbox(output)* method, which adds one given output to the output sandbox of the job.
- *add_property(key, value)* method, which enables developer to add any property, which is required in the JDL file of the job, with name *key* and given value.

The *get_output* method takes the directory, where job output is stored, as a string and returns the result of the Grid Operation. Below a simple wrapper for ping execution on EGEE testbed is presented:

```
class LcgPing
 def ping_submit(host)
   jobSpec = JobSpec.new
   jobSpec.executable='/bin/ping'
   jobSpec.arguments= '-c 5 ' + host
   jobSpec.stdoutput='sample.out'
   jobSpec.stderr='sample.err'
   jobSpec.add_to_output_sandbox('sample.out')
   jobSpec.add_to_output_sandbox('sample.err')
   #jobSpec.add_to_input_sandbox(input)
   return jobSpec
 end

 def ping_get_output(outputDir)
   result = ''
   IO.foreach(outputDir + '/sample.out'){|line|
     result << line
   }
   return result
 end
end
```

For more information on using EGEE testbed and JDL please refer to [55].

Wrappers naming convention Wrapper class name must be identical to the name of the Grid Object (excluding the package name). For instance, the name of a wrapping class for *regadb.RegaAlignment* is *RegaAlignment* and for *cyfronet.gridspace.gem.Namd* is *Namd.*

C.3. Extending a registry

A registry is responsible for storing information about any accessible resources in the ViroLab environment. What is more, it hides resources technologies complexity from the user. That is why if new technology is added to the Grid Operation Invoker, a registry has to be extended by adding support for this technology. Since the GOI system is customizable (for information on customizing the GOI please refer to A.5) and can cooperate with various registries, the registry that is being used must support new technology.

In case of using the Local Registry, which has the technology information hard-coded, only an extra entry is required. A Ruby *Hash*, describing the instance published in new middleware technology, should be added to the collection of instances. If the external registry, the Grid Resource Registry, is used, please refer to the section *Extending Grid Resource Registry* of this online manual [56].

Appendix D

Technology information stored in a registry

In this chapter technology information describing Grid Object Instances in discussed. It's structure and semantics is explained. These sections are usefull for those who:

- *implement adapter for a new technology*
- *wish to have in depth understanding of technical mechanisms of the Grid Operation Invoker system*
- *want to describe some external instances that will be used in an experiment or will be registered*
- *develop new registry or registry client to be used by the Grid Operation Invoker.*

D.1. Technology information semantics

Every *Grid Object Instance* is described by technology information in terms of communication protocol it uses, exposed interface as well as inputs and outputs. Thus the Grid Operation Invoker is able to determine and load the adapter appropriate for the given instance and transparently invoke Grid Operations.

Various middleware technologies require distinct data, but there is a set of common information for all technologies. Table D.1 presents the technology information which is shared by all technologies and which is specific for a concrete middleware.

Technology	Key	Meaning
Common	instId	unique identification of instance
	name	name of the instance
	type	name of the middleware technology
	endpoint	communication address if applicable or nil
	codebase	codebase URL
	method#x	operation of the instance, x is the number of operation, x>=0
	in#x#y	input parameter y of operation x, y>=0
	out#x#y	output y of operation x, y>=0
Web Service	wsType	determines the type of Web Service: RPC or DOCUMENT
	namespace	name space of the RPC Web Service
MOCCA	componentClassName	the name of the MOCCA component class
	portClassName	the name of the port class that is used to interface the component
	portName	the name of the port that is used to interface the component
LOCAL_GEM (WTS or LCG)	subtype	determines the underlying technology: WTS or LCG
	script	Ruby script that is responsible for mapping inputs and outputs of operations

Table D.1. Technology information: common and specific for a concrete technology

D.2. Data structure

Technology information used by the Grid Operation Invoker is a Ruby Hash, which contains pairs of String type. Such solution is very flexible, therefore adding support for new technology does not imply modification in the code of core part of Grid Operation Invoker, nor in registry and scheduler clients.

The code snippet presented below provides a sample technology information about the *EchoService*.

```
techInfo = {'instId' -> '5', 'name' -> 'instance1',
            'type' -> 'WS',
            'endpoint' -> 'http://virolab.cyfronet.pl:18080',
            'codebase' -> 'http:/sample.com',
            'method#0' -> 'echo',
            'in#0#0' -> 'echoString', 'out#0#0' -> 'echoReturn',
            'wsType' -> 'RPC',
            'namespace' -> 'http://virolab.cyfronet.pl/echo'
           }
```

Appendix E

Publications

Universal Grid Client: Grid Operation Invoker

Tomasz Bartyński[1,2] , Maciej Malawski[1,2] , Tomasz Gubała[2,3] , Marian Bubak[1,2]

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

[2] Academic Computer Centre CYFRONET, Nawojki 11,30-950 Kraków, Poland

[3] Section Computational Science,University of Amsterdam, Kruislaan 403, 1098 SJ Amsterdam

In proceedings of the Seventh International Conference On Parallel Processing and Applied Mathematics, PPAM 2007, LNCS 4967.

Invocation of Grid Operations in the ViroLab Virtual Laboratory

Tomasz Bartyński[2] , Maciej Malawski[1] , Marian Bubak[1,2]

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

[2] Academic Computer Centre CYFRONET, Nawojki 11,30-950 Kraków, Poland

In proceeding of the Cracow Grid Workshop 15-17 October 2007, CGW07.

A Tool for Building Collaborative Applications by Invocation of Grid Operations

Maciej Malawski[1,2] , Tomasz Bartyński[2] ,Marian Bubak[1,2]

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

[2] Academic Computer Centre CYFRONET, Nawojki 11,30-950 Kraków, Poland

Submitted for the ICCS08 conference.

Bibliography

[1] ViroLab, 2006. http://virolab.org.

[2] Peter M.A. Sloot, Alfredo Tirado-Ramos, Ilkay Altintas, Marian Bubak, and Charles Boucher. From molecule to man: Decision support in individualized e-health. *Computer*, 39(11):40–46, 2006.

[3] Ian Foster, Carl Kesselman, and Steven Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[4] Web Service Architecture, 2004. http://www.w3.org/TR/ws-arch/.

[5] OASIS Web Service Resource Framwork, 2006. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf.

[6] Ruby home, 2007. http://www.ruby-lang.org/.

[7] The Perl directory, 2007. http://www.perl.org/.

[8] Python programming language, 2007. http://www.python.org/.

[9] Maciej Malawski, Marian Bubak, Michal Placek, Dawid Kurzyniec, and Vaidy Sunderam. Experiments with distributed component computing across grid boundaries. In *Proceedings of HPC-GECO/COMPFRAME Workshop in Conjunction with HPDC'06*, pages 109–116, 2006.

[10] LCG project, 2006. http://www.cern.ch/lcg.

[11] glite - Lighweight Middleware for Grid Computing, 2007. http://glite.web.cern.ch/glite/.

[12] UNICORE, 2004. http://www.unicore.org.

[13] SOAP version 1.2 W3C Recommendation, 2003. http://www.w3.org/TR/soap12-part0/.

[14] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy Sunderam, and Aleksander Slomiński. RMIX: A multiprotocol RMI framework for java. In *Proc. of the Intl. Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 140–146, Nice, France, April 2003. IEEE Computer Society.

[15] eXtensible Markup Language (XML), 2007. http://www.w3.org/XML/.

[16] Web Service Description Language (WSDL) 1.1 W3C note, 2001. http://www.w3.org/TR/wsdl.

[17] Globus Toolkit, 2007. http://www.globus.org/toolkit/.

[18] Apache WSRF project, 2005. http://ws.apache.org/wsrf/wsrf.html.

[19] WSRF::Lite, 2007. http://www.sve.man.ac.uk/Research/AtoZ/ILCT.

[20] Python Core, 2007. http://dev.globus.org/wiki/Python_Core.

[21] WSRF.NET, 2006. http://www.cs.virginia.edu/ gsw2c/wsrf.net.html.

[22] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1999.

[23] T. Priol C. Perez and A. Ribes. A parallel CORBA component model for numerical code coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417 429, 2003.

[24] ProActive - Parallel, Distributed, multi-threaded solutions, 2007. http://proactive.inria.fr/.

[25] Fractal component model, 2007. http://fractal.objectweb.org/.

[26] MOCCA homepage, 2007. http://www.icsr.agh.edu.pl/mambo/mocca.

[27] The Common Component Architecture Forum, 2004. http://www.cca-forum.org/.

[28] Dawid Kurzyniec, Tomasz Wrzosek, Dominik Drzewiecki, and Vaidy Sunderam. Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters*, 13(2):273–290, 2003.

[29] Vaidy Sunderam and Dawid Kurzyniec. Lightweight self-organizing frameworks for metacomputing. In *The 11th International Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, 2002.

[30] EGEE homepage, 2006. http://public.eu-egee.org/.

[31] DEISA project, 2006. http://deisa.org.

[32] Grid Job Submission and Monitoring Web Service, 2007. http://gridsam.sourceforge.net/.

[33] Witty Services, 2007. http://wts.sf.net.

[34] Kepler poject, 2004. http://www.kepler-project.org/.

[35] The triana project, 2003. http://www.trianacode.org/.

[36] Tomasz Gubała, Daniel Harężlak, Marian Bubak, and Maciej Malawski. Semantic composition of scientific workflows based on the petri nets formalism. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 12, Washington, DC, USA, 2006. IEEE Computer Society.

[37] Grid Application Toolkit, 2004. http://www.gridlab.org/WorkPackages/wp-1/.

[38] NetSolve/GridSolve, 2006. http://icl.cs.utk.edu/netsolve/.

[39] Web Service Invocation Framework, 2006. http://ws.apache.org/wsif/.

[40] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby - The Pragmatic Programmer's Guide, Second Edition.* The Pragmatic Programmers, 2004.

[41] Tomasz Bartyński, Maciej Malawski, and Marian Bubak. Invocation of Grid Operations in the ViroLab Virtual Laboratory. In *Cracow Grid Workshop, CGW'07, October 15–17, 2007*, 2008.

[42] Ruby Patterns, 2007. http://www.rubypatterns.org/.

[43] Marian Bubak, Tomasz Gubała, Maciej Malawski, Marek Kasztelnik, Tomasz Bartyński, and Piotr Nowakowski. Virtual laboratory in virolab. In *Cracow Grid Workshop, CGW'06, October 15–18, 2006*, 2007.

[44] ViroLab Virtual Laboratory, 2008. http://virolab.cyfronet.pl.

[45] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques.* Morgan Kaufmann, San Francisco, 2005.

[46] Attribute-Relation File Format, 2002. http://www.cs.waikato.ac.nz/ ml/weka/arff.html.

[47] Grid Resource Registry, 2007. http://virolab.cyfronet.pl/trac/vlvl/wiki/GrrDesign.

[48] Maciej Malawski, Joanna Kocot, Eryk Ciepiela, and Marian Bubak. Optimization of Application Execution in the Virolab Virtual Laboratory. In *Cracow Grid Workshop, CGW'07, October 15–17, 2007*, 2008.

[49] Shibboleth Project - Internet2 Middleware, 2007. http://shibboleth.internet2.edu/.

[50] JRuby download page, 2007. http://dist.codehaus.org/jruby/.

[51] Java SE downloads, 2007. http://java.sun.com/javase/downloads/.

[52] Apache Maven Project, 2007. http://maven.apache.org/.

[53] GOI download page, 2007. http://virolab.cyfronet.pl/ tomek/msc/software/.

[54] EDG documentation, 2004. http://marianne.in2p3.fr/datagrid/documentation/.

[55] LCG tutorial, 2005. https://edms.cern.ch/file/454439/2/LCG-2-UserGuide.pdf.

[56] Adding support for new technologies in the ViroLab Virtual Laboratory, 2007. http://virolab.cyfronet.pl/trac/vlvl/wiki/AddingSupportForNewTechnologies.