Services Lifecycle Management using Distributed Computing Infrastructures in Neuroinformatics

Javier Rojas Balderrama

UNIVERSITE NICE SOPHIA ANTIPOLIS
Ecole Doctorale de Sciences et Technologies
de l'Information et de la Communication

THESE

pour obtenir le titre de
Docteur en Sciences
de l'Université Nice Sophia Antipolis
Discipline : Informatique

présentée et soutenue par
Javier Rojas Balderrama
le 11 avril 2012

GESTION DU CYCLE DE VIE DE SERVICES DEPLOYES SUR UNE INFRASTRUCTURE DE CALCUL DISTRIBUEE EN NEUROINFORMATIQUE

Thése dirigée par Johan Montagnat et Diane Lingrand
et préparée au sein du laboratoire I3S, équipe modalis

Jury :

Bernard Gibaud	Inserm, Rennes	Rapporteur
Johan Montagnat	CNRS, Sophia Antipolis	Directeur
Christian Pérez	Inria, Lyon	Rapporteur
Michel Riveill	Université Nice Sophia Antipolis	President

Services Lifecycle Management Using Distributed Computing Infrastructures in Neuroinformatics

Contents

Abstract

There is an increasing interest among scientific communities for sharing data and applications in order to support research and foster collaborations. Interdisciplinary domains like neurosciences are particularly eager of solutions providing computing power to achieve large–scale experimentation. Despite all progresses made in this regard, several challenges related to interoperability, and scalability of Distributed Computing Infrastructures are not completely resolved though. They face permanent evolution of technologies, complexity associated to the adoption of production environments, and low reliability of these infrastructures at runtime.

This work proposes the modeling and implementation of a service–oriented framework for the execution of scientific applications on Distributed Computing Infrastructures taking advantage of High Throughput Computing facilities. The model includes a specification for description of command–line applications; a bridge to merge service–oriented architectures with Global computing; and the efficient use of local resources and scaling. A reference implementation is proposed to demonstrate the feasibility of the approach. It shows its relevance in the context of two application–driven research projects executing large experiment campaign on distributed resources. The framework is an alternative to existing solutions that are often limited to execution consideration only, as it enables the management of legacy codes as services and takes into account their complete lifecycle. Furthermore, the service–oriented approach helps designing scientific workflows which are used as a flexible and way of describing application composed with multiple services.

The approach proposed is evaluated both qualitatively and quantitatively using concrete applications in the area of neuroimaging analysis. The qualitative experiments are based on the optimization of specificity and sensibility of the brain segmentation tools used in the analysis of Magnetic Resonance Images of patient affected by Multiple Sclerosis. On the other hand, quantitative experiments deal with speedup and latency measured during the execution of longitudinal brain atrophy detection in patients impaired by Alzheimer's disease.

Abbreviations

ACD	Ajax Command Definition
ADNI	The Alzheimer's Disease Neuroimaging Initiative
API	Application Programming Interface
BPEL	Business Process Execution Language
CE	Computing element
CLI	Command–lined Interface
CNS	Central nervous system
CORBA	Common Object Request Broker Architecture
CSF	Cerebro–spinal fluid
CT	Computer tomography
DCI	Distributed Computing Infrastructure
DIET	Distributed Interactive Engineering Toolbox
DIRAC	Distributed Infrastructure with Remote Agent Control
EGI	European Grid Initiative
EMBOSS	The European Molecular Biology Open Software Suite
FLAIR	Fluid attenuated inversion recovery sequence
FSL	FMRIB Software Library
GASW	Generic Application Service Wrapper
GC	Grid Computing
GEMLCA	Grid Execution Management for Legacy Code Applications
GM	Gray matter
GridFTP	Grid File Transfer Protocol
gRAVI	The Grid Remote Application Virtualisation Interface
GUI	Graphical User Interface
GWENDIA	Grid Workflow Efficient Enactment for Data Intensive Applications
HTC	High Throughput Computing
IDL	Interface Description Language
IQR	Interquartile range
JAXB	Java Architecture for XML Binding
JAX-RPC	The Java API for XML–based RPC
JAX-WS	The Java API for XML Web services

JDL	Job Description Language
JSAGA	Java implementation for the Simple API for Grid Applications
JSON	JavaScript Object Notation
LB	Logging and Bookkeeping
LCC	Local coefficient criteria
LCID	Legacy Code Interface Description
MC	Metacomputing
MAD	Median Absolute Deviation
MNI	Montreal Neurological Institute
MRI	Magnetic Resonance Imaging
NMR	Nuclear Magnetic Resonance
OGSA	Open Grid Services Architecture
PD	Proton density–weighted
PET	Positron Emission Tomography
PVE	Partial volume effect
RELAX-NG	REgular LAnguage for XML Next Generation
RPC	Remote procedure call
SE	Storage element
SCUFL	Simple Conceptual Unified Flow Language
SHIWA	SHaring Interoperable Workflows for large–scale scientific simulations on Available DCIs initiative
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SSD	Sum of squared difference
SPECT	Single Photon Emission Computed Tomography
URL	Uniform resource locator
URI	Uniform resource identifier
VIP	The Virtual Imaging Platform
WM	White matter
WMS	Workload management system
WS	Web service
WSDL	Web service Description Language
WSRF	Web services Resource Framework
XML	Extensible Markup Language

Thesis Statement

Introduction

There is a high interest among scientific communities for the creation and investigation of theoretical models, and the acquisition and analysis of experimental data. These activities are fundamental in order to validate previous results, encourage new interpretations, foster collaborations, and enable further findings. However, many important challenges, ranging from infrastructure support to frameworks development passing through modeling, need to be faced to pursue experimental campaigns.

Distributed computing infrastructures (DCI) have become a strong driver for scientific innovation because they enable scientists to mutualize resources such as data, computing facilities, and data analysis procedures. These infrastructures pave the way to the emergence of cross–institutional scientific communities. In addition, DCIs deliver computing and data storage capability needed to address "big science" challenges. To support longstanding experimental campaigns, DCIs provide experiment management frameworks with abstraction layers that shield the users from the complexity of the underlying technologies and tools. Large–scale experimentation on such infrastructures remains difficult to setup and conduct though. Distributing computations related to an experimental campaign, deploying application components over an infrastructure, and monitoring applications executions massively distributed are activities requiring skills that most scientists are not acquainted with. Therefore the amount of work that scientists put to make those infrastructures suited for their specific interest can be considerable and this work rarely addresses the need of a wide range of users in this context.

In practice, data analysis tools developed by scientists make intensive use of command–line interfaced (CLI) applications. The CLI tools are broadly adopted in scientific computation both for practical and historical reasons. They frequently represent legacy validated implementations. The CLI applications provide simple but versatile invocation interfaces, and they are commonly available. However, these applications do not usually follow any standard specification to implement their invocation interfaces. They are not designed to interact with other applications in order to build complex data analysis procedures in the form of assembled processing pipelines, or workflows. Additionally, the profusion of invocation specifications and the lack of a uniform way to process data and generate results make their use limited in a distributed environment. These considerations restrict CLI applications sharing across institutions, even if their users belong to the same scientific community.

Distributed infrastructures serve CLIs through broadly–adopted batch systems. Batches were initially designed to exploit command–line interfaces in remote tools invocation on local or wide-area distributed infrastructures. An extended variety of batch systems and workload management systems have been developed to match the need of different infrastructure scales. Yet, these simple remote computing environments process computing tasks on–the–fly and they do not address the problems of application tools deployment nor consider the structure of experiment workflows.

The tools deployment, and data to process in DCIs involves to handle aspects bound to access and operation of infrastructures, and applications installation. Production infrastructures do not ensure systematically resilient and reliable services. These infrastructures face well–identified problems such as high latency, unfair balance between execution tasks, and a non–negligible failure rate. Furthermore, the deployment is often managed at an administrative level, limiting the autonomy of scientists throughout experimentation.

Efficient scientific experimentation also requires strategies for tools execution and advanced data manipulation. Despite the progresses made in data–intensive application composition with the settlement of scientific workflows, this approach stumbles on integration obstacles for the enactment of applications. Control structures and iteration strategies, that scientific workflows languages provide to automate workflows execution, cannot be fully exploited whereas the parameters of CLI applications are not properly described and the results conveniently handled.

The neuroimaging community, for instance, needs to analyze large brain image datasets. Scientists are interested in designing complex workflows combining image analysis tools from different sources. Their activities are often focused on statistical analysis procedures which involve the processing of large population data. Besides, they deal with sensitive data geographically distributed. The case studies of this community draw attention to the management of applications, their composition and enactment as scientific workflows, and fulfillment of external concerns.

In fact, the instrumentation of scientific experiments is not only conditioned to the infrastructure and the methods that scientists use to conduct their studies. They regularly have to contemplate additional concerns during experimentation to fulfill institutional policies, protect information, or monitor the execution platform. Thus, the integration of external concerns such as data access control, users authentication, infrastructure load balancing, or processing validation is necessary all along the process in order to carry out a successful experimentation.

Nowadays, multiple barriers slow advances in applied research because of the lack of appropriate working frameworks. Scientists usually have access to local computing resources that commonly reach power calculation limits during large experimental campaigns. Although scientists may also count on distributed computing facilities, hence the combination of local and remote heterogeneous platforms makes experimentation challenging. The existing platforms partially allow users to manage the deployment of applications on DCIs, design workflows, and perform efficient executions in a comprehensive environment.

Objectives

As a result of the analysis drafted in the previous section the following objectives motivate the work reported in this manuscript. They aim at improving grid computing experience, specially for neuroimaging scientists, by simplifying experiments design, and by providing efficient enactment.

1. Specify an abstraction of CLI applications supporting executions on DCIs.
2. Implement an extensible framework enabling integration of non–functional concerns.
3. Adopt a deployment strategy ensuring scalable and flexible experimentation.
4. Ensure reliable and resilient executions on heterogeneous infrastructures.
5. Design scientific workflows instances of neuroimaging use–cases.
6. Enact scientific workflows efficiently exploiting local and remote computing resources.

Fulfilling these objectives lead to a technical distributed–computing framework implementation. Exploiting this framework, as part of an interdisciplinary collaboration, the applicative part of this work also focuses on two neuroimaging use–cases: the automatic brain segmentation, and the longitudinal atrophy detection in Alzheimer's disease. These research efforts aim at helping in the treatment of brain conditions encompassing tools and techniques for analysis, modeling and simulation. The establishment of links between interdisciplinary domains encourages new scientific insights, makes existing tools and data more valuable, and new studies more reliable and reproducible.

Contributions

This work is positioned in the field of neuroinformatics, as it deals with the organization of neuroscience applications and data with computational models and their implementations. However, it is not limited to that particular area since the contributions are valid for any domain interested in software interoperability or production DCIs utilization. This work specifically pays attention to the way users handle CLI applications in order to perform large–scale experimentation.

The specification of a scheme to describe CLI applications detailed in Chapter 2 eases their (re)use. The details of the interface invocation presented in a uniform manner becomes an expressive instrument to work with the data structures and types associated to the application's parameters. This abstraction enables the endorsement of applications descriptions to open and compliant implementations of SOA like Web services by transforming instances of the CLI application scheme into standard definitions. Additionally, the introduction of the application's characteristics related to the execution into the definition of the application, such as system requisites or environment dependencies, leads to the adoption of heterogeneous infrastructures and multiple platforms. These characteristics identify clearly the requirement for each execution instance.

The exhibition of CLI applications as modular and interoperable components involves the instrumentation of the logic concerning the interpretation of the arguments, dependencies configuration, and the execution. This transformation is complex due to the need to respect the applications nature, and the integration of non–functional concerns associated to platform of execution or data management. Chapter 3 presents a reference implementation of a SOA approach detailing the lifecycle of CLI applications as services.

The deployment of CLI applications as Web services also brings together flexibility and scalability. The software development involved in this work carries out the implementation of an end-to–end framework for applications wrapping, deployment and execution control. This strategy of deployment engages the manifest replication of services and load balancing of hosting servers in order to achieve large–scale experimentation. In addition, the extensive use of distributed computing infrastructures is granted seamlessly. The framework offers a single interface to process services invocations by dynamic reallocation of resources. The execution resilience and reliability on such infrastructures is ensured by the implementation of resubmission mechanisms, data replication, integration of multilevel job scheduling, and control of remote computing sites.

Finally, applications in neuroimaging are proposed in the second part of this thesis. The representation of CLI applications as Web services allows their composition as scientific workflows. This approach describes the interactions and dependencies of applications using a highly expressive language that includes iteration strategies for advanced data manipulation. The design of scientific workflows promotes efficient executions of data–intensive applications. A workflow enactment involves multiple levels of parallelism delegating at the same time data staging to execution endpoints. In addition, efficiency is enhanced introducing local resources for executions. This inclusion, conceived as a modeling to dispatch execution request based on resources availability, becomes a vehicle to overcome the latency caused by batch systems overheads. It is also a mechanism to reduce the failure rate associated to remote executions.

Context

This work was motivated by and applied to driver projects: NeuroLOG in the area of neurosinformatics, and VIP concerning the improvement aspects of the large–scale experimentation. Both develop collaborative platforms and are grounded on a translational research view. They make extensive use of CLI applications and scientific workflows. The NeuroLOG project constitutes the base context in terms of the reference framework development and use–cases, whereas that the VIP project provides elements to ensure enactment efficiency.

The NeuroLOG project

NeuroLOG[1] is an applied research project that aims at integrating process, data, and knowledge in neuroimaging [Montagnat, 2011]. The NeuroLOG project fosters the adoption of health–grids in a pre–clinical community for supporting multi–centric studies targeting the treatment of four pathologies: multiple sclerosis, brain tumors, cerebral strokes, and Alzheimer's disease.

[1] NeuroLOG project: http://neurolog.i3s.unice.fr/

The NeuroLOG project has developed a platform in a distributed environment. The platform interfaces existing neuroinformatics resources (databases and processing tools) without modification of the legacy environment implemented at each participating site. The platform is grounded on a common domain ontology also developed within the project, which provides a reference for unifying the heterogeneous data representations adopted by the federation of partners. In addition to the federation of resources, NeuroLOG provides an interface to distributed computing infrastructures. It enables the bundling of neuroimage processing and their relocation for remote execution when handling compute intensive tasks. The platform includes a workflow manager used to describe complex image analysis pipelines, potentially involving large datasets. Federated datasets and processing tools are semantically annotated. This domain knowledge is used to validate the coherency of planed processing. The processing is enriched at runtime thanks to the inference of new facts through the application of semantic rules attached to data processing tool classes. The NeuroLOG platform is a prototype deployed over several sites (Grenoble, Paris, Rennes, Sophia Antipolis) to demonstrate the validity of its federated approach.

NeuroLOG is guided by a prospective vision of biomedical research, where data is:

- commonly available to large user communities,
- described and shared using reference domain ontologies,
- browsable through search engines exploiting knowledge represented in ontologies,
- exploitable in an interdisciplinary framework that facilitates the binding of experimental facts from different domains and contexts, and
- applied to heavy–processing tasks implying distributed infrastructures.

Similarly, the processing tools are:

- exposed as Web services for easier dissemination and use,
- integrated in processing pipelines through semantic annotations for easier compatibility validation of their linked inputs and outputs, and
- outsourced to distributed infrastructures such as grids for fast and reliable execution.

The NeuroLOG characteristics, specially those concerning the processing tools, become the requirement analysis of this work. They represent the guiding thread for reuse modeling and development. Additionally, the use–cases considered in this project are suitable examples of study and validation support. In fact, neuroimaging tools are typically computational intensive applications with long execution timespans; work on large–size datasets; and the research teams working with them are specialized on specific topics. Therefore the collaboration and tools sharing not only enhance the understanding of integrative aspects in neurosciences. The conducted studies also provide quantitative information that may be statistically analyzed, corroborated, and compared. The results may be a product of interdisciplinary efforts concentrated at the same time in involved methods and available resources.

The VIP project

The Virtual Imaging Platform (VIP) project[2] targets multi–modality, multi–organ and dynamic (4D) medical images simulation. Integrating proved simulation software of the four main imaging modalities, the platform copes with interoperability challenges among simulators, addresses compatibility issues between organ models and provides transparent access to computing and storage resources.

To tackle interoperability issues, the semantics of models and simulation tools are made explicit. This will be achieved using annotations referring to a set of consistent ontologies describing the organ models, the simulation data processing, the simulation tools and the simulated images. Associated repositories and software interfaces will ease experiment design, assisted simulator and model integration. To address the computational challenge, distributed computing infrastructure technologies are employed. Yet, to cope with reliability issues of large–scale production environments, VIP proposes to develop a multi–infrastructure execution environment able to use both local computing resources (multi–core servers and clusters) and large–scale grids. No heavy code parallelization is involved though: speedup is provided from data and code parallelism only, naturally expressed in simulations.

VIP includes a strong application aspect to guarantee the adequacy of the resulting environment with the needs of imaging techniques developers, model designers and image processing researchers. Specifically, VIP includes four applicative objectives that are used to demonstrate this adequacy. These applications are (1) the validation SINBAD CT simulator, (2) the development of a new US sequence for motion detection, (3) the modelling of inflammation process from MRI simulation, and (4) the evaluation of cardiac segmentation algorithms from multi–modality images.

While the VIP project covers a diversified range of simulations, this work focuses on the reuse and the execution interoperability aspects. Its working environment constitutes an example of the integrative effort of software development, to provide a comprehensive framework to scientists, taking into account non–functional concerns and heterogeneous environments of execution.

Organization

The document is composed of seven chapters, organized in two parts. The first part reports the proposed solution to the challenges identified during the work with CLI applications on distributed computing infrastructures. The second part addresses the scientific workflows as an efficient alternative raised by neuroimaging experimentation at large–scale. Each part may be read independently because it includes the context formulation and the results. However, cross–references throughout the document make a reading thread inviting to follow the sequence of chapters from the introduction to the conclusions.

The first part "Command–line Interface Applications as Services" is organized as follows:

Chapter 1: *Services as building blocks of scientific experiments*. It is a state–of–the–art in the field of services, distributed computing infrastructures, and tools to manage command–line applica-

[2] VIP project: http://vip.creatis.insa-lyon.fr/

tions. This chapter is essential to understand the adopted approach in this work, and identify the technological challenges.

Chapter 2: *Models for reuse of command–line applications*. It represents the theoretical contribution of this work. An encompassing model of service–oriented architecture and global computing is presented. In the same way, a strategy is defined for the efficient use of local resources during workflow enactment. Finally, a complete specification to describe command–line applications is defined. This chapter was partially published in [Rojas Balderrama et al., 2010].

Chapter 3: *Reference implementation framework*. It represents the software development contribution. A comprehensive working environment for wrapping, deployment, and execution of scientific applications is detailed from the lifecycle perspective. This chapter was included in [Ferreira da Silva et al., 2011], and [Rojas Balderrama et al., 2011].

The second part "Scientific Workflows in Neuroimaging" makes extensive use of the development efforts for application reuse, and software integration addressed in the first part. It is organized as follows:

Chapter 4: *Scientific workflows*. It is a comprehensive summary of concepts about data representation, the adopted scientific workflow environment and its underlying language specification. This chapter represents the conceptual pointers required for workflows design and enactment of use–cases introduced in the next chapter.

Chapter 5: *Neuroimaging use–cases*. It reports on the neuroimaging examples adopted in this work: the automatic brain segmentation, and the longitudinal atrophy detection in Alzheimer's disease. This chapter summarizes the interdisciplinary work performed in collaboration with the Asclepios team from INRIA. These use–cases were presented in [Rojas Balderrama et al., 2008], and [Gibaud et al., 2011].

Chapter 6: *Enactment and execution on production distributed computing infrastructures*. It details the scientific workflow instantiation of the neuroimaging use–cases; and the experimentation focused on qualitative results, and execution scalability on production environments. First part of this chapter was published in [Pernod et al., 2008] and the second one in [Rojas Balderrama et al., 2012]

A final chapter recapitulates the conclusions, and states the prospects.

Part I

Command–line Interface Applications as Services

Chapter 1

Services as Building Blocks of Scientific Experiments

Scientific communities take advantage of SOA principles, as such architectures have for long demonstrated their ability to handle interoperability emphasizing on concepts of reusable and autonomous software components (among others). However SOA does not enable legacy applications reuse *per se*. Legacy applications, provided as command lines, are a fundamental part for processing and analyzing scientific data. There are also "new command–line applications" which are not legacy but developed as it for simplicity and the ability to use them though regular batch systems. These applications represent a huge foregoing investment. They encapsulate algorithms that still respond to the expectations of users, and they are not re–implementable using modern techniques due to lack resources and time. In addition, the real need to access such applications from other computers, and the requirement of a programmatic way to invoke them, motivate the implementation of non–intrusive approaches to wrap them as services to enable their reuse. The impact of intensive use of legacy applications also pushed to look for new environments to obtain fast and reliable frameworks adopting distributed computing infrastructures and preserve those applications as the building blocks of scientific experiments.

The command–line interface tools, conceived to be executed on console terminals, are considered legacy applications because they have a simple interface to interact with users and they are not designed to interoperate with other applications or be executed remotely. These applications are executed using parameters to provide inputs, and describe outputs in a syntax that is not always uniform. They depend on system environment variables, and often require additional libraries to run. Web services can be a solution to reuse command–line applications providing a technology stack to deliver results over Internet without worrying on installation or configuration because they can run remotely exchanging information, and interoperating by means of standard mechanisms.

This chapter summarizes the state–of–the–art for the reuse of legacy applications as services, the adoption of distributed computing infrastructures to overcome the increasing need of power computation, and a brief review of initiatives that take into account the SOA principles in legacy application transformations. It introduces several technologies involved in reusability, interoperability, and scalability.

1.1 Interoperable applications

There has been a long–standing desire in software engineering for a standard way of collecting and using software. Initially conventional middleware were developed at a time where the systems were limited to local private networks or conditioned to the Internet using mere adaptions. They were conceived to resolve specific problems in a well–defined context. Later, the interactions between applications, specially when they are located in a distributed environment, were assured by communication models. Protocols of communication were defined standing out expressiveness, convenient combination, and semantic soundness. Technologies implementing such protocols enabled interoperability between programming languages, operative systems, and computer architectures. Two major examples are the Remote Procedure Call (RPC) systems, and the object request brokers.

1.1.1 Remote procedure call

The original goal of remote procedure call (RPC) [Birell and Nelson, 1984] was to provide a transparent way to call procedures in remote computers based on the client/server architecture. In fact, the RPC mechanism is the underlying principle of most of current middleware because it introduced the concept of Interface Definition Language (IDL). An IDL is an abstraction of the procedures representation specifying the input/output parameters. The IDL represents the description of the service provided by the server. Once the IDL is defined, a compiler generates the client and server *stubs*. The stubs are a model of data representation and the references to their implementation. When a client performs a RPC call, the client stubs are used to ask the execution of the remote procedure. Next, the server stubs call the procedure itself and send back the results to the client stub. Finally, the client stub returns the results of the application to the client.

1.1.2 Object request broker

An object request broker [CORBA, 2008] is a middleware supporting the interoperability between remote *objects*. This is a natural evolution of the RPC for adapting to the Object Oriented Paradigm. The goal of this brokers is close to the RPC. They mask the complexity behind the remote invocation. The most famous implementation is CORBA of the Object Management Group.[1] It is a specification and an architecture for the creation and management of distributed and object–oriented applications over a network. The CORBA specification is independent to any implementation regarding the programming language or the operative system. CORBA defines the communication interfaces using IDL–CORBA, a more powerful language compared to RPC because it supports concepts of heritage and polymorphism. Moreover, CORBA implements a dynamic invocation based on a discovery mechanism on the client side that is not possible with RPC. Other types of object brokers are also available, such as DCOM and its descendant COM+ of Microsoft,[2] and the Java Remote Method Invocation[3] (RMI). However DCOM and COM+ are specific to Microsoft operating systems, and RMI is a

[1] Object Management Group: http://www.omg.org/
[2] Microsoft Component Object Model Technologies: http://www.microsoft.com/com
[3] Java RMI: http://download.oracle.com/javase/6/docs/technotes/guides/rmi

Java–based technology restricted to work from the Java Virtual Machine.

1.1.3 Discussion

The effective reuse of CLI application with RPC or object request brokers is difficult to accomplish due to the code instrumentation. The CORBA protocol, for instance, despite its independence regarding programming languages, requires to implement the objects of the communication interfaces. In consequence, this is not practical for widespread use because working with source code is inevitable.

At the same time, RPC and later the CORBA protocol were incapable to resolve completely interoperability problems such as data interchange and execution autonomy in heterogeneous architectures. Moreover, the consensus of such technologies was limited and alternatives based on XML formats emerged quickly because of their neutral approach. This initially promoted their cohabitation speaking the same vocabulary but later the industry identified Web services as a promising implementation to face interoperability.

1.2 Web services

The notion of service was introduced before the concept of Web service was coined by the Open Group (formerly Open Software Foundation) for the specification of the Distributed Computing Environment standard.[4] However, the acquired importance of the concept is associated to the emergence of Web services. The concept of service is defined as an abstract resource representing the possibilities to perform a task in order to guarantee a given functionality coherent from the point of view of the provider and the client agent. This service must be implemented by a concrete provider. Using the base of the service concept, several definitions tried to specify the concept of Web service (WS), for example Curbera et al. [2001] define it as follows:

> *A Web service is a networked application that is able to interact using standard application–to–application Web protocols over well defined interfaces, and which is described using a standard functional description language.*

Web services describe a distributed computing paradigm that differs from other approaches such as CORBA in its focus on Internet–based standards to address heterogeneous distributed computing. The use of standard technologies reduces problems related to the heterogeneity, and it is the key to facilitate the integration of applications. Even more, Web services provide the necessary support for new architectures such as the Service Component Architecture [SCA, 2007].

Web services may be used for the implementation of the Service–oriented Architecture, but they have to follow all its properties, so other components are needed to complement that architecture. Namely, UDDI is used for publishing the services, WSDL for the description of the service, and SOAP for the protocol of communication.

[4] Distributed Computing Environment: http://www.opengroup.org/dce/

Service–oriented Architecture

The Service–oriented Architecture (SOA) is a set of design principles used during the phases of systems development and integration [Erl, 2005]. A system based on SOA focuses on the requirements defined at strategy level, and business process. SOA is also an architecture of distributed systems based on the concepts of services and characterized by the following properties:

- Standardized service contract. Services adhere to communication agreements, as defined collectively by one or more service–description documents.
- Service loose coupling. Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- Service abstraction. Services hide logic from the outside world beyond descriptions in the service contract.
- Service reusability. Logic is divided into services with the intention of promoting reuse.
- Service autonomy. Services have control over the logic they encapsulate.
- Service statelessness. Services minimize resource consumption by deferring the management of state information when necessary.
- Service discoverability. Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.
- Service composability. Services are effective composition participants, regardless of the size and complexity of the composition.

The SOA services are described using metadata. The provider stores the information of services in a directory. A client agent can discover a service based on specific criteria published on that directory. Then the client uses the stored metadata to exchange messages with the service.

1.2.1 Web Services Description Language

The Web Services Description Language (WSDL) [Christensen et al., 2001] provides a model for describing Web services using an XML format. WSDL splits the abstract functionality from the concrete details of the service instantiation. Basically a WSDL is composed of definitions. Every definition includes interfaces (ports), messages, bindings and services. An interface is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Messages are abstract descriptions of the data being exchanged, and port types are abstract collections of supported operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding, where the operations and messages are then bound to a concrete network protocol and message format. In this way, WSDL describes the public interface to the Web service. Data types of messages are not always defined inside the service description. An

additional XML schema actually may be attached to the service description listing types of operation messages.

WSDL is often used in combination with the Simple Object Access protocol (SOAP). In those cases, any client program connecting to a Web service can read the description to determine available operations. The agents interacting with the Web services uses SOAP messages. The client can then use SOAP to actually call one of the operations listed in the WSDL file and exchange messages with the provider calling the operations declared in the WSDL.

1.2.2 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) [Gudgin et al., 2007] provides a definition of information represented in XML format. It is used to exchange structured information and types in a distributed and decentralized environment. SOAP is an independent protocol, that is not attached to any platform or programming language.

The specification of SOAP establishes a standard message format. It may be used in RPC-like transactions or in document–centric message mechanisms. SOAP facilitates the implementation of synchronous and asynchronous communication models. It defines a structured communication protocol containing protocol headers, an envelope section, headers of the message and its body. The contents of the envelope, headers, and body are not defined by the SOAP specification. They are dependent on the implementation. However, the SOAP specification defines how to use those elements. SOAP is not tied to any transfer protocol. It may be used with several protocols such as HTTP or SMTP.

1.2.3 Universal Description Discovery and Integration

The Universal Description Discovery and Integration (UDDI) [UDDI, 2004] is a platform–independent XML–based registry designed to be queried using SOAP messages. It also provides access to WSDL documents describing the protocol bindings and message formats required to interact with the Web services listed in its directory. UDDI is conceptually a catalog server of names and addresses. The information stored by this registry is oriented to human interpretation. For this reason a dynamic binding is not possible because an automated client is not capable to discover a service an build a communication message at execution time.

A UDDI business registration consists of three components (i) White Pages, giving information about the business supplying the service (ii) Yellow Pages, providing a classification of the service or business, based on standard taxonomies, and (iii) Green Pages used to describe how to access a WS, with information on the service bindings.

1.2.4 Summary

Web services were conceived to ease interoperability based on standard technologies, reducing the heterogeneity and providing support for the integration of applications. They are the result of the consensus of different specifications, namely the W3C[5] recommendations the WSDL and SOAP

[5]World Wide Web Consortium: http://www.w3c.org/

[Christensen et al., 2001; Gudgin et al., 2007]. In principle, the reuse of scientific applications is effective embracing techniques like wrapping of the CLI applications in WS interfaces. This non–intrusive approach exposes the applications as services hiding the implementations details, and exhibiting the functionality.

Although the Web community developed the SOA to tackle the challenge raised by software reuse/distribution, it put a little effort in improving performance of SOA–based applications. Meanwhile, the metacomputing and global computing models were developed revolutionizing the access to large–scale infrastructures for optimizing the execution and for promoting the sharing of storage an power calculation.

1.3 Metacomputing and global computing

Distributed Computing Infrastructures (DCI) have become a strong driver for scientific innovation. The increasing need for computing power and data federation arising in many international consortia pushed forward the development of unprecedented large–scale infrastructures. High Throughput Computing (HTC) environments obtained community attention because they deliver large amounts of processing capacity over long periods of time. The way to the development of HTC [Thain et al., 2005] was already paved in the nineties, when the outstanding growth of the Internet in terms of size, reliability, and bandwidth enabled super–computing capability using large amount of regular computing resources geographically distributed. At that time different architectures were defined proposing the base elements of the current implementations. From those architectures two models, metacomputing and global computing, encompass most of concepts related to efficient computation and distributed location.

Computing resources transparently available to the user through networks have been called a metacomputer [Smarr and Catlett, 1992]. Metacomputers are network of heterogeneous, computational resources linked by software. Indeed, to achieve this level of organization efficiently, compute resources must be integrated into a seamless resource that can be easily managed within one framework.

Metacomputing adds another dimension to the configuration management over a potentially arbitrary collection of heterogeneous resources. The framework must be able to identify available resources, acquire any such resource, initialize the computation on it, and eventually terminate. A metacomputing framework must be able to manage resource effectively not only exploiting different machines but supporting different types of parallelism, managing both synchronous and asynchronous control flow among compute nodes, allowing control–oriented and data–oriented synchronization, and managing data locality in order to minimize communication and latency. Communication among compute nodes must be controllable by the application to manage the available bandwidth and tolerance. A metacomputing system allows applications to assemble and use collection of resources on demand, independently from their physical location.

Metacomputing has much in common with both distributed and parallel systems, yet also differs from these architectures in important ways [Foster and Kesselman, 1997]. Like distributed systems, metacomputing must integrate resources of widely varying capabilities, connected by po-

tentially unreliable networks and often located in different administrative domains. However, the need for high performance can require programming models and interfaces fundamentally different from those used in distributed systems. Metacomputing applications, as in parallel computing, often need to schedule communications carefully to meet performance requirements but the heterogeneous and dynamic nature of metacomputing systems limits the applicability of some parallel computing tools.

The resource management for metacomputing is typically based on *resource brokers* [Czajkowski et al., 1998], involved in servicing single requests. They translate application requirements in more concrete resource requirements with the assistance of an information service. The information service is responsible for providing efficient and pervasive access to information about current availability and capacity of resource. At low level the management enables remote monitoring and execution of processes or *jobs* created in response to a resource request, and periodically updates the information service with the current activity.

The metacomputing model was successfully implemented in production environments like the gLite[6] middleware providing access to batch systems. However this approach has some drawbacks. It is inefficient, wasting computational resources while waiting for requests, and it needs to integrate significant mechanisms, techniques and tools to assure allocation of resources, scheduling, authentication, and authorization.

Almost at the same time, the global computing model emerged as a simple and effective abstraction layer to shield the users from the complexity of underlying distributed systems [Foster, 2005b]. Global computing refers to computation over infrastructures available globally, provides uniform services with variable guarantees for security, reliability, scalability, and self–management with particular regard to the programmability of these services. In fact, the adoption of SOA and the subsequent use of Web standards defined a more specific model based on services. The global computing vision requires protocols that are not only open and general–purpose but also standard. Standards allow institutions to establish resource–sharing arrangements dynamically. Those standards are also important as a mean of enabling general–purpose services and tools.

Global computing provides the foundations for the development of large–scale general–purpose computer systems that have dependably predictable behavior for the needs of different organizations. It might be designed to support resource sharing, or services transactions. In essence, global computing is not just middleware, but goes up to software engineering methods. Furthermore, it addresses a range of issues such as mobility, ubiquity, and interactivity.

1.3.1 Grid computing

The *Grid computing* term [Baker et al., 2002; Schwiegelshohn et al., 2010] was adopted to cover the technologies addressing high performance computing in an heterogeneous environment operated by cross–institutional and global–scale initiatives. Almost immediately, the need to interoperate ever more heterogeneous resources led to a paradigm shift.

[6] gLite: `http://glite.cern.ch/`

Grid computing concerns essentially a range of middleware technologies intended to support resource sharing between groups of computers as virtual organizations (VO), a dynamic set of individual and/or institutions defined around a set of rules and conditions [Foster, 2001]. Originally, the research associated to grids was meant to increase computing power by sharing tasks between computers. The essence of Grid computing can be summarized in a system that coordinates resources that are not subject to centralized control, delivering non–trivial qualities of service. The strengths of grids include the security architecture and resource management, conversely identified weaknesses are lack of fault–tolerance and self–management.

1.3.2 Grid services

Beyond harnessing computing power, Grid infrastructures provide a flexible and adaptive support compatible with modern application development methodologies. A convergence took place between Grid technologies and the Web technologies with the Open Grid Services Architecture (OGSA). This architecture represents an evolution towards a Grid system architecture based on SOA concepts and technologies. Grid services are presented as an extended version of Web services that combines specificness of the architecture such as security and decentralization. Their use resulted in the redesign of grids middleware as collections of collaborative services and the emergence of the WSRF standard [WSRF, 2006]. On the practical side, after years of experience and refinement, the Globus Toolkit[7] (GT) produced a widely used de facto reference implementation.

The Grid services [Foster et al., 2002] are improved Web services that introduce statefulness, service data, notification mechanisms, groups of services, lifecycle management, and a more powerful addressing scheme called Grid Service Handle. The addressing scheme proposed by the Grid services implementation uses WSDL as reference format to provide the information about communication. They are based on the Web Service Resource Framework (WSRF), a standardized architecture to submit jobs to the Grid organized on virtual organizations for resource sharing. Grid services also represent the foundations of CLI application reuse on a distributed environment because many frameworks implement them to enable the execution of legacy application as services (see section 1.4).

1.3.3 GridRPC

Among existing middleware and application programming approaches, the Remote Procedure Call over the Grid, or GridRPC model, was developed to ease Grid programming [Seymour et al., 2002]. GridRPC services enable the distributed execution of applications and serve as a communication layer and an invocation interface for high–level software components. It fills the gap between services provided on Grid infrastructures and the programming–level abstraction required to implement a distributed middleware. This approach is also close to SOA because it defines a model where a service is registered in a registry and a client invokes the service on the server [Nakada et al., 2007].

[7] Globus Toolkit: `http://www.globus.org/toolkit`

GridRPC focuses on the invocation of remote procedures across a network rather than on a stack of technologies. For instance, an environment based on independent services involves the setup of all technologies associated to each service (i.e., configuration, execution, monitoring). In some cases, those services share technologies such as a Web server, database management system, etc. However in other cases, the technologies should coexist in the same system independently. In contrast, GridRPC provides a common interface to perform all the invocations, such as file transfer or job submissions. GridRPC also preserves performance rather than adopting a protocol based on XML documents, promoting the direct use of an API. In the same way, GridRPC avoids the reuse of object technologies like CORBA for several reasons, among witch IDL expressiveness that does not specify non–functional requirements, to focus on a simple lightweight implementation that meets the needs of scientific computing [Tanaka et al., 2004].

The adoption of GridRPC promotes interoperability on DCIs without imposing an implementation in contrast to the Globus Toolkit. However both approaches require the source code instrumentation under a reference middleware. Such approaches are, most of the time, designed to access one type of infrastructure at a time, or they apply when components exclusively allow the execution of applications on distributed environments ignoring more complex scenarios like the integration of local resources and heterogeneous platforms.

XtreemOS: Integrated Support of Grid–based Services

In grid computing, physical devices, applications and datasets could all be seen as services. Yet, they are not considered as a technological commodity due to the complexity associated to the management of resources. Initiatives as XtreemOS[8] aim at resolving transparency, scalability, interoperability, and security issues from the user point of view. XtreemOS organizes the access to the available resources in Virtual Organizations as an integrated support on top of an operative system. The software architecture of the platform distinguishes two main layers the XtreemOS-F, and the XtreemOS-G one [Coppola et al., 2008]. They provide, respectively, local support and integration of different resources into a single computing platform.

While initiatives as Grid Services or GridRPC have been build to resolve the access to Grid environments using intermediate middlewares, XtreemOS proposes an approach where an underlying operative system is extended for enabling and facilitating Grid computing. Middlewares like Globus toolkit have been adopted where institutions agree on a reference implementation to commission the operation of the infrastructure. In the same way, the principle of transparent access of XtreemOS is valid when the same operative system is available everywhere. Nevertheless, neither of these approaches can be adopted on infrastructures composed by heterogeneous service sources or operative systems like the European Grid Initiative. Most of the EGI services (i.e., storage, execution, monitoring) are based on common middleware components, and the resources share the same system configuration, however the autonomy of each organization imposes to harness services according to the provided interfaces independently.

[8] XtreemOS: http://www.xtreemos.org

1.3.4 Summary

Grid computing, resulting from the evolution of metacomputing and global computing models, addresses the permanent need of computing power. To achieve that goal, institutions supply resources, develop middleware, define specifications, charter rules of use, and foster the development of tools supporting the existing middleware. For instance, the utilization of batch systems in the execution of CLI applications were commonly adopted on production grids by means of a uniform job description language. Several projects implemented strategies on top of middlewares to improve the access to the distributed computing infrastructures. Those initiatives were interested in providing tools letting users execute their applications efficiently on the DCIs. Specifically, they have been paying attention to implement non–intrusive tools for reusing and executing CLI applications because technologies and services such as the Grid services do not pay attention in fine–level management of applications or they only resolve partially concerns going from interoperability to usability. Some of these efforts are detailed in next section.

1.4 Survey of tools supporting command–line applications reuse

The idea of software reuse is not new [Rich and Walters, 1983]. Several approaches have been studied such as the reuse at programming level to build applications directly from several pieces of code [Bigot et al., 2008] or approaches of reusing executable command–line applications directly [Mateos et al., 2008]. The reuse of such applications still involves important effort in the scientific community because this facilitates the integration of a wide range of applications in current research. There are several toolkits and environments which use a non–invasive approaches to wrap the command–line applications as services. They do not create new binary executables or modify the existing CLI applications. The concept of service in these environments, in most cases associated with SOA, creates an opportunity for reusing such applications under different circumstances depending on the target infrastructure, the protocols of communication or the domain of application, among others. In this section some relevant examples are presented. This work does not pretend to be exhaustive, focusing only on recognized initiatives that have proven to be useful and provide at the same time interesting concepts of service–oriented design.

1.4.1 LONI Pipeline

The LONI Pipeline [Dinov et al., 2009] is a graphical environment for construction, validation, and execution of neuroimaging data analysis applications. It is a packaged solution to allow distributed infrastructures utilization, to facilitate data provenance, and to provide a significant library of computational tools including automated data format conversion. As part of its environment, one of the tools facilitates the integration of heterogeneous applications as *modules*. These modules represents well–defined standalone applications, comprising local or remote binary executables and services with well–defined command–line syntax. The modules are created providing general information like authors, version, name, description, and detailed information about the syntax of the parameters. The parameters may be directories, enumerations, files, numbers, strings or flow

controls that provide a sense of data typing support to the modules. The modules may also include dependencies to the applications. The LONI Pipeline integrates the modules in pipelines involving large number of datasets and multiple processing tools.

Data in terms of databases, data services, and file systems, along side with the modules may be integrated in the environment. This flexibility of integration permits efficient resource management. The LONI Pipeline environment is focused on neuroimaging data and analysis protocols. However, by design, it is domain agnostic and its architecture may be used where computationally intense tasks are performed. Unfortunately, LONI Pipeline modules are not constructed on standardized bases and the catalog of services is only useful within the environment. Although the integration to broader distributed infrastructures is supported with the adoption of its Distributed Pipeline Server (DPS).

1.4.2 GASW

The Generic Application Service Wrapper (GASW) [Glatard et al., 2006a] is a dynamic service which aims at enabling the execution of CLI applications as services at runtime. This service is generic, wrapping an application behind its standard interface, and submitting a job instance to a DCI. The GASW service simplifies the embedding of applications into services interpreting a description of the application. Its Legacy Code Descriptor is an XML–based file which contains the name and location of the executable, the access method of the input data, the command–line options of the parameters, and the name and access method of the libraries or scripts that may be needed for the execution aside from the target binary executable. The GASW service leverages external middleware submission methods for the execution. It also implements application grouping service calls to optimize the execution time.

GASW does not address the deployment of the applications though. The generic service is not data typed and does not have a high–level interface to create the descriptions. This later issue makes the wrapping of applications difficult because it requires in–depth knowledge of the ad–hoc XML structure and technical concepts associated to the distributed infrastructures. The service exposition used on GASW is based on a factory pattern. This process involves to use a generic interface to dynamically process the applications arguments. That kind of optimization can be considered harmful in a contract–first approach because clients trying to execute the original service may be unable to find it as well defined service loosing qualitative information regarding the types or nature of parameters.

1.4.3 GEMLCA

The Grid Execution Management for Legacy Code Architecture (GEMLCA) [Delaitre et al., 2005] is a general architecture for deploying CLI applications as Grid services without the need for code modification. GEMLCA aims at providing an infrastructure to deploy applications as OGSA–compliant services. Its architecture is composed of four basic components:

1. The *Compute Server* represents hardware resources, such as a single computer or clusters, on

which applications in form of binary executables are potentially available to make them accessible through Grid services.

2. The *Grid Host Environment* implements a service–oriented Grid layer on top of a compliant middleware. This environment connects the Computer Server with a grid. Current distributions of GEMLCA support OGSA–built Grid services based on the Globus Toolkit.

3. The *GEMLCA Resource* provides a set of Grid services (i.e., code factory and processor) which exposes applications as services. Along with the Grid Host Environment, the GEMLCA Resource is installed on the Compute Server representing a GEMLCA Grid Resource.

4. The *GEMLCA Client* comprises the client–side software. There are two types of GEMLCA Clients: (1) a command–line interface, installed on any machine through which a user would like access to the GEMLCA resources, and (2) a Web portal based on GridSphere[9] to provide a graphical interface through which a user can access to the applications as Grid services.

The GEMLCA Resource is responsible for hiding the native nature of an application by wrapping it with a Grid service, and processing service requests coming from users. The deployment of such a service implies that the application may run in its native environment on a Compute Server. The GEMLCA Resource handles the application using an XML–based Legacy Code Interface Description (LCID) file. This file provides metadata about the application, such as the executable path, the job manager, the execution environment, and information parameters including name, type, order, regular expressions for input validation, etc.

GEMLCA does not require coding modification of applications and the effort from clients is minimized using the graphical interface. In spite of this, the deployment process of new Grid services involves administrative tasks on the server side. Since the current GEMLCA has GT2, GT4, and gLite submitters, applications can be executed/submitted to all machines with these middleware. GEMLCA uses stateful services primary to support a multi–user environment, but not for the lifecycle management of CLI applications though.

1.4.4 gRAVI

The Grid Remote Application Virtualisation Interface (GRAVI) [Chard et al., 2009] is a plug–in extension to the Introduce toolkit [Hastings et al., 2007]. This toolkit is designed to support the Grid service development through three identified steps:

1. The *creation*. The developer describes at highest level basic attributes about the service such as name and namespace. The implementation of the service is then created with these configuration properties using the Introduce engine.

2. The *modification*. The developer adds, removes or modifies service methods, properties, resources, and security configuration. In this step, strong typed service interfaces are created using pre–registered and well–defined schemas. The Introduce toolkit includes the notion of data repositories that maps defined types to application input parameters.

[9] GridSphere portal framework: http://www.gridsphere.org/

3. The *deployment*. The developer deploys the service to a Grid service container after the specification of deployment and security configuration. A deployment component gathers the libraries required for the service as well as those files which contain the actual runtime code of the service.

gRAVI allows users to wrap binary applications as secure WSRF–compliant services without requiring to write any implementation code, description files, or deployment scripts. This plug–in extends the Introduce toolkit creation and modification steps adding new backend code creation processes and complementary graphical interface within the Introduce Graphical Development Environment. This interface removes the need for users to run scripts or create/modify description files. gRAVI services offer synchronous and asynchronous invocation methods. They also include methods to stage data with several encoding formats including GridFTP and base 64 encoded binary data. Each service also exposes interfaces to monitor the running application via polling or notifications.

gRAVI provides a way to reduce the cost of creating Grid services by simplifying the development and deployment process. Nevertheless, the processing of parameters is based on existing service type schemas that are not trivial to create and requires the management of repositories. Despite its interesting features, the design of gRAVI forces to use in each service all its dependencies increasing significantly the packaging size to detriment of library reuse so this method is not advisable for a large number of services. Additionally, it supports exclusively execution on distributed infrastructure, banishing the potential of light or short–term executions on local servers.

1.4.5 Soaplab2

Soaplab2 [Senger et al., 2008] is a framework that allows service providers to make command–line applications accessible as Web services. It is based on metadata descriptions of the programs that includes information about the description, type, provider, names and types of input data or command–line parameters, and names and types of resulting output. Soaplab2 uses ACD, a format originally created by the European Molecular Biology Open Software Suite[10] (EMBOSS) to define in a uniform way bioinformatics tools, to create the descriptions, and to transform them in a corresponding XML format file. The XML description is stored and used by the Soaplab2 server to execute the applications processing the input data and to retrieve the results. The architecture of Soaplab2 server can optionally use local databases for keeping results persistently. On the client side, Soaplab2 provides a rich Web–based interface, which allows users to select a service, specify its inputs, start the service, and display the results. Soaplab2 has a rich client library supporting an extensible protocol layer to assure interoperability with different Web service standards.

Soaplab2 can automatically generate and deploy Web services on top of existing command–line applications. It is especially suited for applications with well described input and output parameters allowing integration of applications within a single programming interface. Nevertheless, Soaplab2 does not resolve the build/install/deploy cycle because it uses development tools such as

[10] EMBOSS: http://www.emboss.org/

Ant[11] and Maven[12] to perform these tasks. Soaplab2 does not provide an interface to wrap the applications as Web services. Thus the work directly with the ACD format is a cumbersome requirement due to its technical characteristics. Furthermore, Soaplab2 is not oriented to execute the application on distributed infrastructures.

1.4.6 Comparison

The main goal of developing tools like LONI Pipeline, GASW, GEMLCA, GRAVI, or Soaplab2 is to provide robust and extensible frameworks enabling efficient utilization of resources, and to provide the means for dissemination and validation of research protocols and scientific innovation. An important experience has been gained with their development [Krishnan and Bhatia, 2009]. From this experience common elements are identified to underline their potentialities and remaining issues:

- Descriptions of applications. The characterization of CLI applications is necessary for binding the application to a service interface. The elements defined in the definition must fulfill fine–grained details of arguments, including types, and the target infrastructures.
- Interoperability. Interoperability is assured by open protocols. Web services (and their Grid variants) mechanisms for describing, accessing, and securing services provide the shared vocabulary. The value of an exposed service is measured with regards to the capability to discover, and access it.
- Scalability. The data volumes and computational demands are often beyond the capacity of a centralized server. Distributed infrastructures, like clusters and grids, and the interconnection with the services are suitable solutions to respond this challenge.
- Usability. The control of the middleware hosting services and their lifecycle management play and important role in the use and dissemination of scientific applications. This control does not have to limit users throughout their experimentation by incorporating complex technological layers of administration.

From the interoperability point of view, most of the initiatives (with the exception of LONI Pipeline and GASW) have recognized the undisputed need of Web protocols. These protocols defines data types and the interfaces of operations. GEMLCA and GRAVI, work directly during creation of services to provide the WSRF–based services. This alternative manages security concerns in parallel to the functional requirements. Although the implementation behind the resulting services depend on the installation of the adopted middleware (i.e., Globus Toolkit). On the other hand, Soaplab2 is focused only on basic standard protocol profiles.

Concerning the scalability, all approaches described but Soaplab2 take into account distributed infrastructures to delegate the execution of the applications. This adoption has natural advantages

[11] Apache Ant: http://ant.apache.org/
[12] Apache Maven: http://maven.apache.org/

over limited computing resources but includes a side–effect. All tools adopting scalability measures focus on the execution by delegation ignoring the potential benefit of nixed execution both on local servers and distributed infrastructures.

Usability is covered at different levels. GASW, only covers the execution of applications but it does not manage the creation and deployment stages leaving to the user the role of manually creating the description, deploying the application, and invoking the resulting service. GEMLCA and Soaplab2 includes interfaces for the invocation (graphical or through API) but again the creation and deployment are left as administrative tasks. GRAVI is more careful providing a complete chain from creation to invocation as well as LONI Pipeline that hides completely the administrative tasks.

Interoperability, scalability, and usability are the desirable properties required to wrap applications as services. In order to enable the complete life–cycle management of services and provide them flexible execution mechanisms, features such as the execution on local resources; high–level interfaces to create and enable a programmatic invocation; and a public and well–defined scheme for the description of executions are expected. Since none of the reviewed frameworks provide all these characteristics (see Table 1.1) their adoption is not possible due to the lack of features such as local execution, client APIs, etc. In the same way, extension is difficult because implementation incompatibilities with execution middlewares, or copyright limitations.

	Loni Pipeline	GASW	GEMLCA	gRAVI	Soaplab2
Interoperability			✓	✓	✓
Scalability	✓	✓	✓	✓	
Usability	✓		✓ˣ	✓	✓ˣ
Local execution	✓ˣ	✓			✓
Client API				✓	✓
Graphical UIs	✓		✓ˣ	✓	
Public schema	✓ˣ				✓

Table 1.1: On top of the table three high–level properties required from tools supporting command–line applications to wrap CLI applications as services are identified. On the bottom, the expected technical features are presented. The tick (✓) represents availability of the feature, and the partial check mark (✓ˣ) shows the tool only covers such feature partially.

1.5 Conclusion

In this chapter a concise state of the art on service–oriented approaches dealing with legacy applications was presented. The evolution of applications interoperability and software reuse was described across the chapter taking services as reference. These services represent an invaluable mean of reusing algorithm implementations resulting from years of research. In fact, they become the building blocks of scientific experimentation because their combination may result in a com-

plete processing pipeline. Specifically, the services are created from CLI applications by wrapping them using SOA interfaces. This approach, followed by several initiatives, not only pursue remote invocation. The efficient execution on distributed environments is also considered.

Three main features were identified to create a fully functional framework for the generation of such services: the interoperability, scalability, and usability. The review presented here shows some remaining issues that are not completely addressed by the available solutions. This makes obvious the need of a new approach that provides a complete lifecycle of services support and an improved description of CLI application that integrates all the execution details. Therefore, in the following chapter a conceptual contribution to resolve the current implementation gaps of existing tools is proposed.

Chapter 2

Models for Efficient Reuse of CLI Applications

The Grid community has almost unanimously adopted the Service–oriented principles introduced in chapter 1. The integration of Global computing with SOA has been largely addressed with the adoption of implementations such as the Globus toolkit [Foster, 2005b]. In this approach services for monitoring resources, discovery and management, security, and file management are providing a complete stack of technologies in an integrated environment. Nevertheless, the use of such technologies is not always possible. The infrastructures based on heterogeneous middleware, like production grids, do not ensure directly the implementation of Grid services–based on the global computing model, because they are conditioned to the general adoption of a common underlying technology. In those infrastructures the interface to resources should be integrated through a non–intrusive component, and the standard interconnection should be enforced with a modular implementation connected to the target infrastructure.

This chapter presents three models for reusing of CLI applications and taking advantage of local resources efficiently. First, a non–intrusive *hybrid model* merging global computing with SOA to enable the adoption of a service–based framework in production DCIs. This model aims both at taking advantage of the SOA principles and the existing distributed computing infrastructures. Second, a *model for efficient use of local resources* combining the adoption of DCIs. This approach is defined to dispatch jobs for local execution or for submission to a DCI based on the execution behavior of services. Finally, a *model using metadata to describe command–line applications* expose them as services and enable their execution. This description includes a fine–grained management of input/output data structures, dependencies, and execution environment, in order to resolve and use the described metadata at runtime. These three models together enable the efficient execution of CLI applications in order to (re)use them as services.

2.1 Enabling SOA in production Grid infrastructures

This section outlines the existing gap between the SOA approach and the global computing model, identifying the weaknesses of each one, and proposes a practical solution to bridge them together.

In a global computing environment, clients connect to a brokering service that handles the requests on their behalf as illustrated in Figure 2.1. The broker has extensive knowledge on the dis-

tributed resources available on the infrastructure. It selects the resource that can best handle each client request and delegates its actual execution. The applications to be executed are transported from clients to the broker and then to the computing resources. All managed resources in a global computing model are allocated temporarily to each computation task and have minimal system requirements. The broker may act as a proxy caching the requests and corresponding results if the clients disconnects for a given time. This model is very efficient to control and balance the overall system workload and therefore addresses well the needs of High Throughput Computing over long periods. The broker also implements a scheduler and/or resource allocator that optimizes the usage of the system.

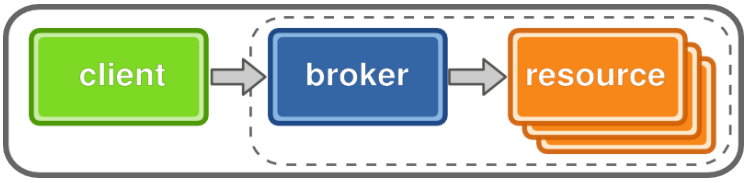


Figure 2.1: Global computing model

Global computing implementations such as Grid infrastructures typically serve CLIs executions through batch processing. Batch processing involves the execution of a series of programs or "jobs" dispatching them to distributed resources. Batch systems process CLI applications which do not required user's interactions. These applications provide versatile invocation interfaces that can be interpreted on–the–fly when a job invocation is sent to the broker. All input data is set as command–line parameters or an equivalent file representation like the Job Description Language (JDL). In this operating environment a program processes the data automatically, and produces a set of output data files. Despite their long history, batch applications are still critical in most organizations in large part because many core business processes are inherently batch–oriented (i.e., data are collected into batches of files and are processed in batches by the program). Most workload management systems use batch processing to maximize usage because (i) batch systems allows sharing of computer resources among many users and programs; (ii) they shift the time of job processing scheduling large amounts of tasks; and (iii) batch systems avoid idling the computing resources with manual intervention and supervision. Nevertheless, each implementation defines their job invocation methods, thus the interoperability in global computing environments is restricted to managed infrastructures.

Conversely, in a traditional SOA framework, services embedding the business logic are pre–deployed over a set of resources and invoked remotely though a standardized interface. Clients perform direct connection and invocation to the services as illustrated in Figure 2.2. Before reaching the target(s) service(s), clients have to query a registry service, that at least provides business services localization information and possibly implement workload management strategies.

Interoperability is granted by the standard interfaces and protocols inherited from SOA implementations. Defined message channels decrease the complexity of applications, shading light on functionality rather than communication. However, this model requires to instrument the business logic with a service interface, and pre–deployed applications over the computing resources.

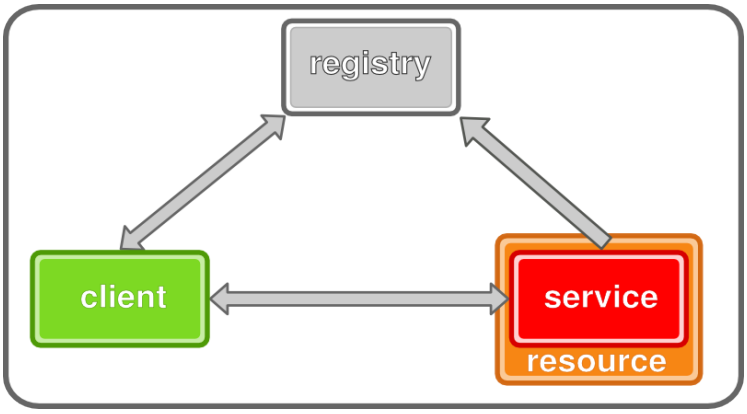


Figure 2.2: Service Oriented Architecture

Management of large–scale applications implies complex and frequent deployment procedures. Furthermore, clients are directly exposed to the communication with various resources and therefore they need to integrate complex concerns related to scalability, performance, reliability, fault-tolerance, security, etc.

To tackle the issues of both approaches a new model is presented proposing the convergence between global computing and SOA as illustrated in Figure 2.3. This solution enables dynamic allocation of resources encompassing the properties of SOA with the interfaces provided with global computing implementations. The approach integrates the need of intensive computing infrastructures using standard interfaces of communication. The cycle of deployment and executions should be integrally taken into account to ensure invocation of services without affecting the internal architecture of any infrastructure. Clients use SOA mechanisms for execution requests before deploying dynamically the services. Then the broker processes those requests as regular tasks and return results to clients using the same messaging paths. To describe the tasks delivered to the broker, the service has to adapt services messages into a language interpreted by the broker. In this work, we will consider the transformation to JDLs defining the CLI invocation.

The dynamic allocation of resources permits the execution of services directly on the deployment point, or the delegation to a broker for execution on remote resources based on the execution needs and the work load. There is not intervention of the client for the task dispatch after the execution request becoming a transparent resolution of resource allocation. Each element of the hybrid model performs its tasks as an independent module, while coordinating message transfers to provide final results.

The hybrid model proposed here provides an alternative to execute applications as services by transforming the execution interfaces from broker dispatching to interoperable messages. This transformation uses the model of Section 2.3 because that definition enables the interpretation of executions associating input arguments to the commands, and matching the results. That definition also provides the information concerning the artifacts and system environment required for the allocation and execution. Nevertheless, this model does not provide any mechanism to address salient issues of production DCIs such as latency and high failure rate of execution. In fact, the use

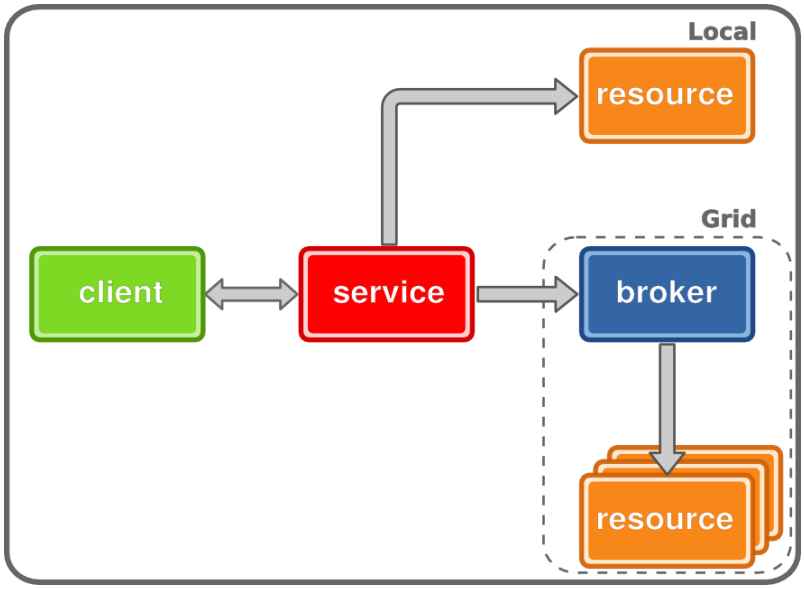


Figure 2.3: Hybrid model merging global computing and SOA

of local resources (e.g., mainframes or private clusters) increases the probability of success full execution. Normally such resources are synonym of reliability in terms of dedicated and permanent access. In the next section a complementary model is defined to combine local resources with DCIs in the allocation process. However, the proposed model does not take into account aspects like data transfer, or hardware heterogeneity because they are directly associated to the dynamic behavior of a production distributed infrastructure.

2.2 Efficient use of local resources

Executing large–scale applications on DCI faces several well–identified issues often causing poor applications performance (either under–performing execution time or complete application failure). In particular, low reliability, high latency, and unfair balance between job executions are recurrently reported in the literature dealing with large–scale experimentation. The definition of a model for the efficient use resources addresses partially these issues reducing the execution delay when submitting to distributed infrastructures by the introduction of local resources, and managing their load to prevent saturation.

In spite of the large number of computing resources available on DCIs, the waiting time of a job to obtain a computing resource may increase considerably with a big number of jobs simultaneously submitted to the infrastructure. This latency is particularly not negligible for short–execution jobs. The use of local resources for executing applications decreases the number of jobs submitted remotely and therefore reduces the management time of jobs to be processed on the DCI. Furthermore, the application performance is improved since local resources are more reliable and jobs are executed without latency. However, a strategy is required to ensure that local resources are not overloaded when many jobs are executed. In view of this, a decision model is defined below to dis-

patch incoming jobs for local execution or for submission to a DCI relying on execution time and memory consumption. The proposed model makes it easy to switch between the broker and the local resources. The service interface, as shown in Figure 2.3, hides the heterogeneity of computing infrastructures and delegates execution to different kind of resources.

The decision model is based on the composition structure of several services in order to create a complete analysis processing. It makes the assumption that each service i among the k services used $\{i \in \mathbb{Z}^+ \mid i \leqslant k\}$ is consuming a fixed amount of resources when executing (r_i memory space, and t_i execution time). It will also be assumed that the execution DCIs are large enough to handle simultaneously all computation tasks triggered by the invocation of the application services at runtime.

Let R denote the memory consumed on the local resource for all running services including r_j which would be an incoming service of type j executed locally at a given time. The value of R is computed according to Equation 2.1, where n_i denotes the number of services of type i. The volume of assigned memory must not exceed R_{MAX}, the available memory installed on the local resource ($R \leqslant R_{\text{MAX}}$).

$$R = r_j + \sum_{i=1}^{k} n_i \times r_i \tag{2.1}$$

Making the hypothesis that production infrastructures have sufficient computing resources to execute all submitted services, the execution time of a service composition T_{MAX} would be the longest path of its representation as a graph (i.e., the critical path). Therefore, the execution time in the local resources T must be shorter than this theoretical threshold in order to avoid penalizing the final execution time ($T \leqslant T_{\text{MAX}}$). The value of T, as shown in Equation 2.2, represents the sequential execution time of all services running on local resources distributed on all available processor units, where N_{CPU} denotes the number CPU cores.

$$T = \frac{t_j + \sum_{i=1}^{k} n_i \times t_i}{N_{\text{CPU}}} \tag{2.2}$$

Algorithm 1 shows the procedure to decide whether a job is executed locally or submitted remotely. The estimation of R and T is performed each time an incoming service is enacted by the workflow manager. Meanwhile, the value of n_j is updated for accounting.

The need to bridge intensive computing models and SOA principles, and the definition of a strategy for the efficient use of local resources do not solve challenges of rich characterization of applications during the execution, including data types and a uniform structure definition of complex arguments. The adoption of the SOA is only complete with the existence of a clear contract defining the operations and all the involved parameters. Most scientific CLI applications developed in the research community are neither designed to ease the description in a standard way nor to separate systematically the invocation interface from the environment requirements or invocation details. These are the reasons to justify the incorporation of an abstraction of command–line applications in order to transform this definition in a compliant format as WSDL.

Algorithm 1 Dispatching of incoming service (r_j, t_j)

Require: $[r_i]_k$ memory benchmark of services
Require: $[t_i]_k$ execution time benchmark of services
Require: $[n_i]_k$ number of services running by type
Require: R_{MAX}, T_{MAX}, and N_{CPU}

for i in $\{1, \ldots, k\}$ **do**
  $R = R + r_i \times n_i$
  $T = T + t_i \times n_i$
end for
$R = R + r_j$
$T = (T + t_j)/N_{\text{CPU}}$
if $R \leqslant R_{\text{MAX}}$ **and** $T \leqslant T_{\text{MAX}}$ **then**
  service is executed locally
  $n_j = n_j + 1$
else
  service is submitted to a DCI
end if

2.3 Abstraction of command–line applications

A formal description of command–line applications ensures their proper (unambiguous) use. The formal definition promotes portability by characterizing a clear and consistent syntax of common features and constraints. Such a description also grants the exposition of CLI applications as services in an SOA approach by interpreting the description and creating an interface for invocation. It enables the construction of commands at runtime once the description is processed in combination with the invocation input parameters, and it provides the information to draw out the results.

The following of this section details the syntax of a CLI application description based on its metadata as a model represented in an XML schema (the complete schema is listed in Appendix 6.4). A running example is used below to exemplify several characteristics when performing the formal description of CLI applications. In some cases, additional examples are included *in line* to better explain some advanced features of the specification. A typical neuroimaging application is described so the invocation details and execution constrains are clearly explicit. It summarizes some of the characteristics that scientists need to identify and associate with inputs and static parameters for later execution. The choice of a neuroscience application is deliberated because of the strong motivations presented in the introduction chapter. In order to facilitate the reading a distinctive typography convention indicates different connotation in the text.[1]

[1]Emphasized text (*sample*) is used the first time an new concept is introduced. Its explanation usually follows directly after the introduction or it is explicitly defined in a subsection. Bold text (**sample**) is used to refer a definition in the specification. Sometimes these elements are used to describe others before they are introduced, therefore they remain in bold to differentiate from regular content. A monospace typeface (`sample`) indicates the term is an XML element or attribute, it also indicates command examples in the system console. Text written in this type is always related to coding. Sans serif typeface (sample) shows examples of concepts defined in the schema or possible options of an attribute.

Running example

BrainVISA[2] is a software that allows users to trigger sequences of treatments in series of images. The treatments are performed by calls to command–lines. One of these applications for calculation of images is AimsLinearComb. It performs a sum of two brain activation maps. The AimsLinearComb tool performs a linear combination using the formula $I_{1+2} = a * I_1/b + c * I_2/d + e$, obtaining a fusion of two binary functional–analysis activation in form of a new volume. In spite of its apparent simplicity, AimsLinearComb is a good example to show some characteristics of the CLI application schema because of the manipulation of special file formats, implicit arguments or dependencies, and the interpretation of the resulting outputs. An example of the command–line required to execute AimsLinearComb is shown in Figure 2.4, where lwlebge.img and lwdupje.img are the images to combine resulting in the image lwtest.img, and the numerical parameters are used for adjustment. In practice the application execution is complex due to some assumptions the user should know:

- The input and output are Analyze format images. This kind of image consists in two files with the same base name but different extensions, namely IMG and HDR, containing the raw image data and the header metadata respectively. In the command–line however, only the IMG file name appears explicitly representing the image. The file name of the HDR file is inferred from the IMG one.

- The tool execution produces a text file with the extension MINF. This file is not expressed in the command–line but it is part of the results along with the output image in Analyze format.

- AimsLinearComb needs several libraries for standalone execution. The user should configure the environment to include them in the list of the system. In Unix–like systems is possible to add to the `LD_LIBRARY_PATH` environment variable the directory path where those dependencies are located.

Additionally, the user should be able to retrieve the standard output or error messages generated by the tool, so all these concerns have to be considered in the description of the tool.

```
$AimsLinearComb -i lwlebge.img -a 200.0 -b 1.0 \
                -j lwdupje.img -c 20.0 -d 1.0 \
                -e 0.0 -o lwtest.img
```

Figure 2.4: CLI invocation of AimsLinearComb application

The proposed XML grammar for the definition of any CLI application separates the application description from its resources (and system environment parameters) in two parts, to ease execution and deployment [Lacour et al., 2005]: the *interface*, which provides the detailed information

[2]BrainVISA: `http://brainvisa.info`

concerning the invocation of the application; and the *implementations*, which specify all the references to the artifacts associated to the application including its configuration environment. These elements are developed in the following subsections.

2.3.1 Interface

The description of the interface {`interface`} includes all information related to the application and the CLI **arguments**. Additionally, general data is included describing a **version**, **description**, **organization**, **contact address**, a **symbolic name**, **copyright policy**, and **reference** of the application.

2.3.1.1 Service version

The version {`version`} defines a unique number–based schema to state a declaration of a service representing the CLI application. The version is used for keeping track of possible variants of the same tool. Typically this identifier includes three numbers separated by a period: major version, minor version, and a build number, this schema may be arbitrary though. This version is used, in combination with the **symbolic name**, to declare the URL of the Web service, therefore the version must always be set. For example:

1.0.0

2.3.1.2 Service description

The service description {`description`} defines a substantial description of the CLI application. This section does not define any format or extension. It may include a short description of application scope, file formats and conventions, configuration or examples of the command invocation. For example:

AimsLinearComb service performs a sum of two brain activation maps using Analyze file format.

2.3.1.3 Organization

The organization {`organization`} contains the information of the CLI application author, vendor, or distributor. Alternatively it may contain the information of the service builder. For example:

BrainVISA

2.3.1.4 Contact address

The contact address {`contactAddress`} specifies an email, phone, or electronic form of the person in charge to contact in case of feedback about the service is required. For example:

admin@i3s.unice.fr

2.3.1.5 Symbolic name

The symbolic name {`symbolicName`} specifies a unique, short name of the service. This name should be a representative alias of the service making reference to the CLI application. The symbolic name

is used, in combination with the **version**, to declare the URL of the Web service therefore the symbolic name must always be set. As a reference, the format should respect the Java identifier convention.[3] For example:

brain_map_sum

2.3.1.6 Copyright policy

The copyright policy {`copyright`} contains the copyright labeling of the CLI application. If the service is publicly available, its use supposes the respect of the author's copyright. For example:

CeCILL licence version 2

2.3.1.7 Reference

The reference {`reference`} must contain a URL pointing to an external page about the CLI application, or a unique bibliographic identification such as the DOI or the PMID. For example:

http://brainvisa.info/doc/documents-4.0/shfjcommands/commands.html#aims_AimsLinearComb

2.3.1.8 Arguments

The collection of arguments {`arguments`}, contains the information of each application's parameter. They are declared respecting the order of appearance in the command–line. This collection is not required when the application does not include any argument. Otherwise a detailed declaration of parameters should represent syntax of each argument in order to construct the command–line to execute.

A set of attributes defines the nature of the argument, and the details are defined using independent elements. The attributes include: an **identifier**, a **category**, a **data type**, a **mapper**, and boolean attributes to describe the **implicitness** and the presence of a **space**. All these attributes have enumerable values declared explicitly in the schema. The elements enabling the declaration of the argument are: a **label**, an **option**, a **hint**, and the **content** and the **nesting properties**.

Argument identifier The identifier {`identifier`} defines a unique reference to the argument. The procedure that yields the generation of the identifier is based on the MD5 algorithm that generates a fingerprint of the **label** value. For example:

07cc694b9b3fc636710fa08b6922c42b

Category The category {hookup} identifies the stream sense of data used to receive/transmit arguments for/from the application. Arguments should be identified as *input*, *output*, or *constant* (i.e., simple flag) streams. Hence, input arguments become the required parameters to execute the CLI application. The output arguments are the expected results after the execution. Finally, the constant streams are invariant values required to process the command–line but they are not part of inputs nor outputs. The possible values of this attribute are:

[3] Java Code Conventions: `http://java.sun.com/docs/codeconv/CodeConventions.pdf`

- input,
- output, and
- constant

Data type The data type {type} refers to the supported classification of primitive data definitions in the schema. This classification determines the possible values for that type; the operations that can be done on values of that type; and the meaning of the data. All arguments are typed but they are not associated to ranges or machine built–in types. The possible values of this attribute are:

- string,
- integer,
- double (for floating–point numbers), and
- URI (references to files)

Nesting properties The nesting properties {nesting} contains the information to build collection of data as arrays. An array stores a number of elements of the same data type in a specific order. They are accessed using an index to specify which element is required. The array–based definition of the arguments follows the array programming principles [Hellerman, 1964] where data is represented as simple elements or scalars or collection of elements or arrays. Arrays may be nested at any depth (multidimensional). The elements enabling the declaration of the nesting properties are:

- the **dimension** of the array {dimension},
- the **element separator** of arrays {separator},
- the **initial character** {beginCollection} and the **final character** {endCollection} identifying the array scope.

Arrays are typically used to organize complex structures of data representing arguments. For example, it is common in neuroimaging to represent an image as a set of files, where each file corresponds an ordered element of the collection. In that case, the dimension of the nesting properties is set to 1 and so on. The nesting properties are ignored when the **dimension** is equal to zero because it represents a scalar value.

Mapper The mapper {mapper} identifies the source of an argument in order to associate a value to its content respecting the declared data type. The notion of *mapper* is loosely based on the definition found in the Switft system [Zhao et al., 2007]. According to that definition a mapper is responsible of accessing data and converting it to/from a format that conforms the defined types. The possible values of this attribute are:

- console (default for strings, and numbers),
- filesystem (default for URIs),
- pattern, and
- archive

The *console* mapper can be associated to all data types. In this case, the value of the argument should be taken from the standard input by parsing the value when the argument is declared as input. Similarly, the value of the argument should be taken from the standard output when the argument is declared as output transforming the resulting strings in formatted array representations. As convention, each line of the standard output represents a different argument value. For example, the following line represents an three dimensions array of strings for a given output argument retrieved from the standard console:

```
[[[a,b]],[[c]],[[]]]
```

The *file system* and *pattern* mappers can only be associated to the URI data type. For both mappers the value of the argument is a simple file reference when the argument is declared as input. On the other hand, the value of the argument is processed differently when the argument is declared as output. If the mapper is defined as *file system*, the value of the argument is associated directly as a file reference with the name declared in the **content**. This implies that the resulting file exists in the file system. If the mapper is defined as pattern, the value of the argument is processed matching the regular expression declared in the **content**, returning either the first matched file reference or a list of all matched ones. For example, in order to retrieve all files which name begin with a number and have the IMG extension, the **content** of the argument is defined as follow:

```
[0-9].*\.{IMG|img}
```

The *archive* mapper can be associated to all data types. In this case, the value of the argument should be taken from an additional configuration file where the structure of the **content** is defined for inputs and outputs.

Implicitness The implicitness {`implicitness`} identifies if the argument should be interpreted as implicit (hidden) argument in the command–line or if the argument is explicitly declared on it. This boolean attribute may be declared only with URI data types. The value must be true to include a required file that is not declared in the command–line but it is mandatory for execution, or to obtain a resulting file from the execution that is not declared in the command–line. For instance, the text file with the extension MINF introduced in the running example is not defined as argument but it is required for the execution, therefore the declaration of this extension resolves the requirement.

Space The space {`space`} identifies if the value of the argument is preceded by an **option** including a white space in between. The value must be true to include an space before the *content* of the argument.

Label The label {`label`} or logical name specifies a unique, short name of the argument. This name should be a human–readable and representative alias. The label is used to declare the argument of the Web service operation, therefore the label must be set if the argument is declared. As a reference, the format should respect the Java identifier convention. For example:

```
image_1
```

Option The option {option} specifies the preceding flag to the value of the argument. This element is not necessary if the command line does not require it. The value of the option should be set as declared in a terminal console including preceding dashes. For example:

-i

Hint The hint {hint} specifies additional human–readable information of the argument. This element may be used as short description of the argument. It does not have any effect over the command–line or the execution. For example:

First volume image to combine.

Content The content {content} specifies the actual value of the argument. The notion of *content* denotes a special interpretation of its attributes and embedded elements because the metadata of the content should be resolved dynamically in function of the assigned values. For instance, in the running example the content associated to the first argument (-i) denotes a replacement resolution. It means that the base name of the argument is used to resolve the second file component of the image associated with the extension hdr.

A set of attributes defines the nature of the content, and its details are defined using independent elements. The content is defined based on a **resolution model** attribute, or alternatively on a boolean attribute to describe the **template resolution**. Both attributes have enumerable values declared explicitly in the schema. The elements enabling the declaration of the content are: the **matter** and a **list of extensions**.

Resolution model The resolution model {model} identifies the dynamic processing used to resolve the values of the argument based on a reference potentially combined with a list complementary information. This resolution is performed when the content of the argument does not represent directly the actual value used in the command–line. The possible values of this attribute are:

- regular (default),
- directory,
- replace, and
- expand,

The *regular* resolution model can be associated to all data types. In this case, the value of the argument does not require any dynamic resolution. It should be taken directly as parameter of the Web service operation and then as part of the command–line, when the argument is declared as input. If this resolution model is set then the declaration of the **list of extensions** is ignored. For example, the image file lwtest without its header file can be obtained if the resolution is defined as *regular* and the *content* is set to:

lwtest.img

The *directory*, *replacement*, and *expansion* resolution models can only be associated to the URI data type, performing a dynamic resolution to resolve the real file references at file system level.

If the resolution model is defined as *directory*, the value of the argument corresponds to the file reference's file name. The list of files contained in the file reference's file name are expected as real values for the application. If this resolution model is set the **list of extensions** is ignored. For example, when all files resulting from the execution including original binaries are expected as results in a directory the *content* is set to (the dot character implies current working directory in Unix–like operative systems):

.

If the resolution model is defined as *replacement*, the value of the argument corresponds to the file reference's file name (with a base extension). References resulting from the combination of the file reference's file name (without extension) and each declared extension in the **list of extensions** are expected as the actual values for the command–line execution. For example, using the running example the expected files for the first image have the IMG and HDR extensions, but only the IMG file should be declared on the command line, so the *content* is set to:

lwlebge.img

Then the *list of extensions* includes the value:

hdr

If the resolution model is defined as *expansion*, the value of the argument corresponds to the combination of the file reference's file name (without extension) and each extension declared in the **list of extensions**. References resulting from the combination of the file reference's file name (without extension) and each declared extensions in the **list of extensions** are expected as real values for the command–line execution. For example if a given output or input requires several files with the same base file name but different extensions and all files should appear in the final command line, the *content* is set to:

lwlebge

Then the *list of extensions* includes the values:

img,hdr

Although last two resolution models map the same image the argument differs because the resulting values is respectively (assuming other values by default):

lwlebge.img

and

lwlebge.img lwlebge.hdr

Matter The matter {`matter`} contains the raw value of the content. It is represented as a sequence of characters but the actual value is denoted by the data type after the content resolution. For example:

lwtest.img

Extensions list The extensions list {`extensions`} contains a coma–separated collection of file extensions, without extension separator. This element is required for the content resolution when the data type is defined as URI. For example:

img,hdr

2.3.1.9 Interface example

In order to show the resulting metadata description of the running example, its first argument is set in Figure 2.5. In this case the type of the argument with the label "image_1" is a URI. The content value will be set after execution because it represents an input argument. It uses a file system mapper in combination with a replacement resolution model. This enables to manage the HDR file associated to the parameter. This argument is explicitly declared and uses an space between the option "-i". The argument represents a scalar, it means the dimension is 0 (zero) therefore the nesting properties are ignored.

```
<ns1:argument ns1:identifier="07cc694b9b3fc636710fa08b6922c42b"
              ns1:hookup="input"
              ns1:type="URI"
              ns1:mapper="filesystem"
              ns1:implicitness="false"
              ns1:space="true">
   <ns1:label>image_1</ns1:label>
   <ns1:option>-i</ns1:option>
   <ns1:hint></ns1:hint>
   <ns1:content ns1:model="replace" ns1:template="false">
      <ns1:matter></ns1:matter>
      <ns1:extensions>hdr</ns1:extensions>
   </ns1:content>
   <ns1:nesting>
      <ns1:dimension>0</ns1:dimension>
      <ns1:separator>&amp;quot;,&amp;quot;</ns1:separator>
      <ns1:beginCollection>&amp;quot;&amp;quot;</ns1:beginCollection>
      <ns1:endCollection>&amp;quot;&amp;quot;</ns1:endCollection>
   </ns1:nesting>
</ns1:argument>
```

Figure 2.5: Declaration of an application's argument

2.3.2 Implementations

The description of the implementations {`implementations`} includes the configuration variables and the resources of the service that are required to execute the command–line application. This definition includes the concepts of *artifact* and *environment* as part of a hierarchical organization. This organization endorses the reuse of such variables/resources when they are common in the definition hierarchy avoiding, at the same time, repetitive definitions or redundant packaging of files.

The implementations are defined as an **implementation** list. They correspond to different builds of the same application. At least one implementation must be declared as part of the implementations. Each implementation {`implementation`} includes a **release version** {`release`}, a collection of platforms {`platforms`}, a global **environment** {`configuration`}, and a global **artifact** {`attachment`}.

Similarly, the platforms are defined as **platform** lists. They relate to different computing infrastructures. At least one platform must be declared as part of the platforms. Each platform {`platform`}

is defined by an **infrastructure**. The platform includes a collection of profiles {`profiles`}, a shared **environment** {`sharedEnvironment`}, and a shared **artifact** {`sharedArtifact`}.

At the deepest level, the profiles are defined as a **profile** list. They represent different computing architectures. At least one profile must be declared as part of the profiles. Each profile {`profile`} is defined by a **programming model**. The profile includes the **main application** file {`target`}, a bound **environment** {`boundEnvironment`}, and a bound **artifact** {`boundArtifact`}.

The final composition of implementations, platforms and profiles should include at least one **artifact** by branch because it contains the main application representing the service. The resulting alternatives of the composition may be induced from the syntax diagram shown in Figure 2.6.

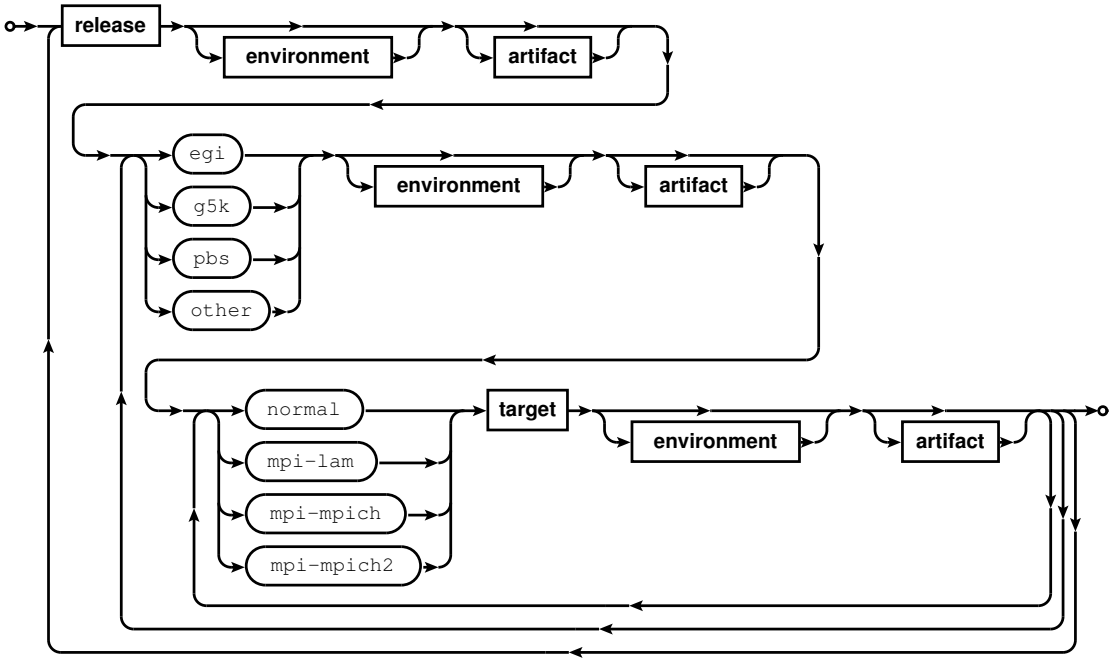


Figure 2.6: Syntax diagram of the implementations. The diagram describes possible paths between elements by going through other non–terminals definitions and the terminals values. Terminals are represented by round boxes while nonterminals are represented by square boxes.

2.3.2.1 Artifact

The artifacts contains a URI to a compressed file reference that includes the resources of the application such as binary files, libraries, and configuration files. For example:

file:///usr/local/share/aims-package.zip

2.3.2.2 Environment

The environment specifies the configuration variable(s) needed for the execution of the application like the PWD variable that represents the current working directory on the Unix–like systems.

2.3.2.3 Infrastructure

The infrastructure {infrastructure} identifies the target computing infrastructure where the application may be executed. The default value implies that the application does not require to be executed on a distributed computing infrastructure. The possible values of this attribute are:

- single (default),
- egi (for the EGI production grid),
- g5k (for the Aladdin/Grid'5000 research cluster),
- pbs (for portable batch systems clusters) and,
- other

2.3.2.4 Programming model

The programming model {job} identifies the implementation required to execute the application. It is associated to the communication protocol in parallel computing. The default value implies that any parallel implementation is required to execute the application. The possible values of this attribute are:

- normal (default),
- mpi-lam,
- mpi-mpich,
- mpi-mpich2,

2.3.2.5 Implementations example

A complete example of the implementation description is shown in Figure 2.7. It declares one release 1.0.0 to be executed on the egi production Grid using the default programming model normal. The executable binary AimslinearComb is set explicitly. All the resources (main binary and system libraries) needed for execution are grouped on the aims-package.zip artifact. Any common or shared artifacts, nor global configuration or shared environment are declared.

2.3.3 Related work

Several schemas for the description of CLI applications are proposed in the literature. For instance, the open software description [van Hoff et al., 1997] was created to distribute applications over the network but it is more oriented to contexts describing hardware dependencies, and ease the automatic installation/upgrade of software components than describing the application's invocations.

Other examples are Soaplab2 using ACD [Senger et al., 2003], and GEMLCA using LCID [Kiss et al., 2005]. These approaches focus on domain–specific applications or well–described input/output parameters. They do not take into account collections of data nor the dependencies associated to the execution of the application. Moreover, the description of implicit parameters or parameters linked to multiple data, cannot be described using those formats. Another example of description with a defined schema is the LONI Pipeline. It includes definitions quite similar to the ones

```
<ns1:implementations>
    <ns1:implementation>
        <ns1:release>1.0.0</ns1:release>
        <ns1:platforms>
            <ns1:platform ns1:infrastructure="egee">
                <ns1:profiles>
                    <ns1:profile ns1:job="normal">
                        <ns1:target>AimsLinearComb</ns1:target>
                        <ns1:boundEnvironment/>
                        <ns1:boundArtifact>file:///aims-package.zip</ns1:boundArtifact>
                    </ns1:profile>
                </ns1:profiles>
            </ns1:platform>
        </ns1:platforms>
    </ns1:implementation>
</ns1:implementations>
```

Figure 2.7: Declaration of the application's implementations

presented in the description of CLI applications. However, the declaration of applications is done in combination with the modules compositions therefore they do not represent independent services. On the other hand, GASW [Glatard et al., 2006a] does not provide a declared schema. The use of ad–hoc formats does not allow users to identify all the features and compare them with others at XML–level because they are interpreted in the business code directly.

Considering the representation of data, the XML Dataset Typing and Mapping (XDTM) [Moreau et al., 2005] is used in SwiftScript to define a mapping between the logical organization of data and their underlying physical structure. It is used to represent files as structured collections but other data types are not considered.

The description of CLI applications has also been considered in Grid computing. The Job Definition Language JDL [WMS–JDL] is used to submit jobs to the Grid describing an application, its parameters, input/output data, etc. The difference with the model presented in this section is that the JDL language specifies instances of execution, not applications metadata.

The model presented in this section is comparable to general approaches like JSON[4] for representing data collections, assuming the natural differences of scope and implementation. First, JSON is not domain–specific, in the sense that it is not defined to describe CLI applications. Otherwise, JSON does not allow to override the separator, or the array identifiers. However, it declares strings using quotes, making possible to differentiate between singleton arrays and empty values. This latter characteristic becomes a feature in the definition of the proposed schema because most of real cases (i.e., Unix–like tools) do not use quotes to describe the arguments in order to represent valid strings. For instance, with JSON an array of strings looks like [“*one*”, “*two*”, “*three*”], using the schema defined in this model, the same array looks like [one,two,three], or <one;two;three>, or even *one two three* (without braces and with spaces as separator). However with JSON the arrays [“”] and [] have different meanings, singleton array and empty array of a given value respectively, but using our proposed schema is only possible to define the second array. Alternatively, JSON or other

[4] JSON: http://json.org/

data serialization formats such as YALM[5] may replace the default convention used in this work to define uniform collections of data, letting the possibility of overriding the nesting properties. This has an important impact in the description of CLI applications because most of them use simple lists separated by spaces to represent simple collections fitting the default command–interface environment.

2.4 Discussion

Three contributions are presented in this chapter in order to enable the efficient reuse of CLI applications. A hybrid approach for the execution of services, a simple model for efficient use of local resources based on composition of services, and a schema for the description of applications.

The hybrid approach provides a bridge between the distributed computing environments and service–oriented architectures, capitalizing on the features of both models. In combination, the resulting approach takes care of intensive computing availability offered by High Throughput Computing environments for efficient executions. It addresses technical challenges respecting open standards and transparency at different levels. However, there is no intention to specify aspects tied to any particular solution. Nor to modify the behavior or fix any defects of CLI applications by improving tolerance to invocation errors or security during execution.

The model incorporating local resources as part of the execution environment along with DCI strengthen the hybrid model because it improves the reliability of production environments and reduce the execution latency by a balanced and scalable integration of servers instances.

The definition of the model to characterize CLI applications is an intermediate layer between the description of *applications* from an operating system point of view [POSIX.1–2008], and the representation of domain–specific knowledge associated to them. In fact, the design of this definition honors SOA specifications like the WSDL of W3C [Christensen et al., 2001]. The resulting CLI application description may be considered as a standard representation.

The models presented in this chapter are a natural evolution of GASW [Glatard et al., 2008]. They represent incremental efforts to reuse CLI applications taking advantage of distributed computing infrastructures and the SOA principles. A fine description of applications is formalized, and the approach to resolve the allocation of resources is described as well. This conceptual contribution is the base of a reference implementation detailed in the chapter 3 that shows the relevance of this approach as a non–intrusive CLI–application wrapper and dynamic re–allocator on distributed infrastructures.

[5] YALM: http://yaml.org/

Chapter 3

Reference Implementation Framework

The Java–based Interoperable Generic Service Application Wrapper framework (*jigsaw*), described in this chapter, is a Reference Implementation of a modular system that implements the models detailed in Chapter 2. It generates services wrapping CLI applications as services. These services have operations to perform the execution of such applications on the host server or on distributed computing infrastructures. The *jigsaw* framework also takes into account the integration of non–functional concerns.

The *jigsaw* framework provides much more than a mere invocation interface to CLI applications. It provides a complete mechanism to package applications and their dependencies into a service artifact. It deploys those artifacts on a server and publish them as standard Web services. As an execution interface, *jigsaw* is also involved with the remote invocation, including files transfer for proper processing of remote resources. *Jigsaw* therefore provides a full range of functionality, making the services autonomous, relocatable and compliant to the target infrastructure for execution. The framework is composed of three independent but complementary modules:

1. An end–user interface for creation and deployment of services
2. A resource allocation engine hosted on a WS container
3. A generic programmatic interface for services invocation

The end–user (graphical and command–lined) interfaces are based on the specification introduced in Section 2.3. Similarly, that specification is used to implement a library set hosted by a service container that works as the engine for dynamic resources allocation. This allocation enables the execution of CLI applications using Web services leveraging the convergence of the global computing and SOA principles described in Section 2.1. Conversely, the API for services invocation represents an independent module in terms of implementation. It enables the standardized and transparent consumption of Web services. These three modules are detailed below following the lifecycle of services in the framework.

3.1 Lifecycle of services

The lifecycle describes the process of creation, deployment, and invocation of services. The *jigsaw* framework defines a service as a unitary bundle. A service is created wrapping the CLI application in an artifact along with its description and all the required resources to execute the application like other binaries or programming scripts, libraries, and configuration files.

Once the service is created, the framework publishes the service deploying the artifact on a services container. An interface based on the description, exposing the application as standard Web service, is generated during this process. At this point, the service is ready to reallocate dynamically all the bundle resources on different computing infrastructures through the *jigsaw* engine configured on the server. The deployment on the services container for the dynamic reallocation represents the implementation of the *hybrid model* introduced in Section 2.1.

The invocation of the operations declared on the service are performed by clients implementing consumers of the Web service. The *jigsaw* framework provides an API to carry out this task. Any Web service contract, derived from the schema defined in Section 2.3, may be interpreted and then invoked with the same methods retrieving the results of the remote execution. A high level view diagram of the whole lifecycle of services in the *jigsaw* framework is shown in Figure 3.1.

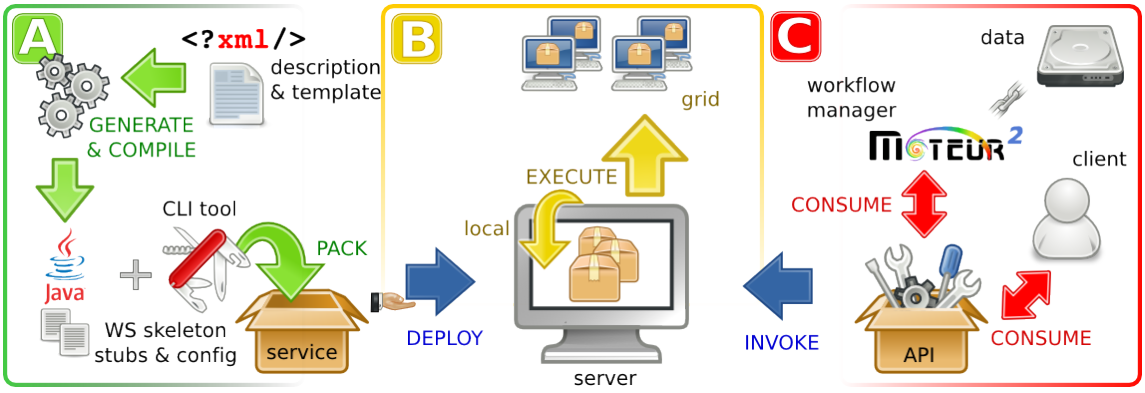


Figure 3.1: Lifecycle of services in the *jigsaw* framework. From left to right: (A) the service is built using a description and generating stubs and configuration files; (B) the service is deployed on a services container ready for dynamic reallocation on DCIs or locally; (C) the invocation of services is granted by a generic API or direct calls using standard WS calls by providing references to the datasets.

3.1.1 Creation and deployment of services

From the end–user point of view, services are created automatically using the graphical interface illustrated in Figure 3.2. The procedure aims at being as simple as possible, filling in the description form that includes the details of all arguments and the execution environment. An artifact representing the service is generated after the description is done. The transformation mechanism of the description into the service interface, and the packaging of resources in the artifact are transparent to users. The resulting file is a portable artifact because it can be deployed on any configured server. Thus users are only aware of the deployment endpoint reusing the artifact conveniently.

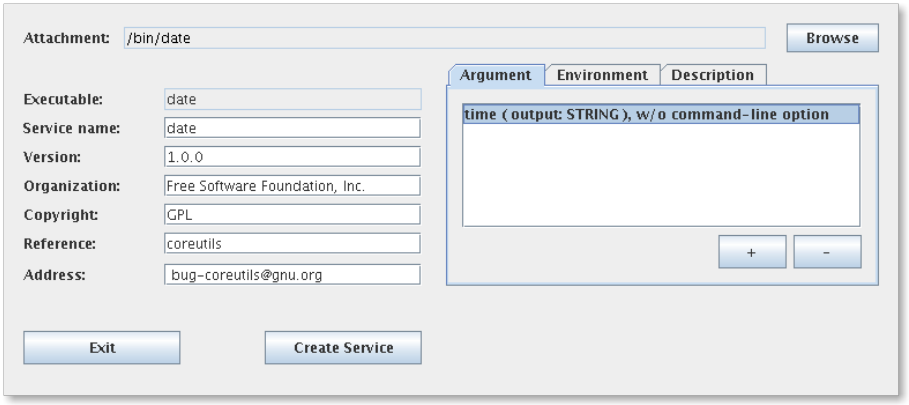


Figure 3.2: GUI of client application for generation of services

The description of the CLI application represents the metadata. This metadata is modeled using the XML schema detailed in Section 2.3. That schema is the starting point to create the service because it provides a data model to express the structure and constrains of the application. The generation of the service consists in transforming the description into a WS interface.

Data binding gives a useful object view of the metadata without losing access to the original information, and delivers performance benefits using unmarshalling and efficient methods to access XML schema build–in data types. JAXB,[1] an open source tool, provides a data binding mechanism by automatically creating a mapping between elements of a XML schema to bind, and the members of a class to be represented as objects in memory. It takes advantage of the richness and features of XML giving a full schema support and the corresponding Java classes. JAXB provides an XML fidelity keeping the full infoset after unmarshalling in an XML instance, and honors schema constraints.

There are two approaches to create Web services: the top–down or "contract first" based on the initial declaration of the WSDL document; and the bottom–up or "implementation first" working with the source code and later generating the WSDL associated to that code. The bottom–up approach is a suitable scenario for the *jigsaw* framework because the service interface may be generated as Java code using the metadata of the application and the data model transformation resulting from the data binding.

Jigsaw implements a scaffolding approach to transform the metadata into source code [Sellink and Verhoef, 2000]. The transformation is based on a template engine that provides sources and the required files to let the server interpret that code after compilation. This approach of dynamic generation of code is required because all service interfaces are customized for each wrapped application. The names and data types of all input arguments detailed in the description are preserved in the resulting Web service description as well as the expected outputs. The generated WSDL declares the CLI application metadata as WS–compliant data types. These types are used in the SOAP messages for the invocation of the service.

Jigsaw internally uses Velocity,[2] an open source tool that defines a simple template language used to create and render documents that format and present a data model as macros. Nonethe-

[1] JAXB: http://jaxb.java.net/
[2] Apache Velocity: http://velocity.apache.org/

```
#macro( varname ) $extra.toLowerCase($argument.type)$argument.label #end
#macro( vartype ) $extra.toJavaType($argument.type) #end
#macro( varspace )
  @WebParam(name = "$argument.label", targetNamespace = "http://i3s.cnrs.fr/jigsaw") #end
#macro( varbracket ) $extra.getBrackets($argument.nesting.dimension) #end

#set($inlength=$extra.getInLength($application))
#set($suffixclassname=$extra.getSuffixClassName($application))
package jigsaw.ws;
⋮
@WebService(serviceName = "$application.getSymbolicName()-$application.version",
  portName = "jigsawPort", name = "jigsaw", targetNamespace = "http://i3s.cnrs.fr/jigsaw")
@SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.WRAPPED,
  style = SOAPBinding.Style.DOCUMENT, use = SOAPBinding.Use.LITERAL)
public class Jigsaw$suffixclassname {
  @Resource private WebServiceContext wsContext;

  @WebMethod(operationName = "local")
  @WebResult(name = "localResult", targetNamespace = "http://i3s.cnrs.fr/jigsaw")
  public JigsawOutput$suffixclassname local(
#if ( $inlength > 0)
    #set($counter=1)
    #foreach($argument in $application.arguments)
        #if ($argument.hookup == "INPUT")
            #if ($counter < $inlength) #varspace() #vartype() #varname()#varbracket(),
            #else
                #varspace() #vartype() #varname()#varbracket())
                throws SOAPException {
            #end
            #set($counter=$counter+1)
        #end
    #end
#else ) throws SOAPException {
#end
      JigsawOutput$suffixclassname output = new JigsawOutput$suffixclassname();
      try {
        Object[] objects = null;
        Description description = DescriptionFactory.getInstance(this);
        ⋮
        output = (JigsawOutput$suffixclassname) activity.fire(new Object[]{
#if ($inlength > 0)
    #set($counter=1)
    #foreach($argument in $application.arguments)
        #if ($argument.hookup == "INPUT")
            #if ($counter < $inlength) #varname(),
            #else #varname()});
            #end
            #set($counter=$counter+1)
        #end
    #end
#else objects});
#end
⋮
```

Figure 3.3: Snippet of the Velocity template used to generate the service skeleton

```
1  package jigsaw.ws;
2  ⋮
3  @WebService(serviceName = "brain_map_sum-1.0.0",portName="jigsawPort",
4     name = "jigsaw", targetNamespace = "http://i3s.cnrs.fr/jigsaw")
5  @SOAPBinding(parameterStyle = SOAPBinding.ParameterStyle.WRAPPED,
6     style = SOAPBinding.Style.DOCUMENT, use = SOAPBinding.Use.LITERAL)
7  public class JigsawBrain_map_sum100 {
8    @Resource
9    private WebServiceContext wsContext;
10
11   @WebMethod(operationName = "local")
12   @WebResult(name = "localResult", targetNamespace = "http://i3s.cnrs.fr/jigsaw")
13   public JigsawOutputBrain_map_sum100 local(
14     @WebParam(name = "image_1", targetNamespace = "http://i3s.cnrs.fr/jigsaw") URI uri_image_1,
15     @WebParam(name = "a", targetNamespace = "http://i3s.cnrs.fr/jigsaw") Double double_a,
16     @WebParam(name = "b", targetNamespace = "http://i3s.cnrs.fr/jigsaw") Double double_b,
17     @WebParam(name = "image_2", targetNamespace = "http://i3s.cnrs.fr/jigsaw") URI uri_image_2,
18     @WebParam(name = "c", targetNamespace = "http://i3s.cnrs.fr/jigsaw") Double double_c,
19     @WebParam(name = "d", targetNamespace = "http://i3s.cnrs.fr/jigsaw") Double double_d,
20     @WebParam(name = "e", targetNamespace = "http://i3s.cnrs.fr/jigsaw") Double double_e,
21     @WebParam(name = "o", targetNamespace = "http://i3s.cnrs.fr/jigsaw") URI uri_o)
22     throws SOAPException {
23       JigsawOutputBrain_map_sum100 output = new JigsawOutputBrain_map_sum100();
24       try {
25         Object[] objects = null;
26         Description description = DescriptionFactory.getInstance(this);
27         ⋮
28         output = (JigsawOutputBrain_map_sum100) activity.fire(new Object[]{ uri_image_1,
29           double_a, double_b, uri_image_2, double_c, double_d, double_e, uri_o});
30 ⋮
```

Figure 3.4: Snippet of the resulting Java source code of the service skeleton

less, the approach described here is valid with any template–based engine. Velocity aims at ensuring a clean separation between the representation and the business tiers using *context* objects. This representation is merged with the template (Figure 3.3 shows an excerpt of the template, details on the implementation are in Appendix 6.4) to produce the source code of the service and the configuration files. The context object is a central concept of the engine. It is the carrier of data between the model representation of the information layer and the template. Since the data model is represented as objects, Velocity makes them directly accessible via the references defined in the template and substitutes the values with the instance of the description (see also Figure 3.4 for the corresponding excerpt of the Java code generated from the template code of Figure 3.3 after substituting object values and processing the macros). The template–based procedure generates:

- a WS interface based on the standard specification of Web services,
- a configuration file of the WS engine, and
- a configuration file of the services container.

Different specifications exist to build Web services from Java code. Some of them are independent efforts such as the Apache Axis implementation,[3] and others are based on standard Java Specification Requests. The latest specification for WS applications and clients is the Java API for XML Web services[4] (JAX-WS). This specification replaces the JAX-RPC API reflecting the move away from RPC–style. JAX-WS represents the "modern" Java SOAP implementation of Web services making extensive use of the annotations mechanism introduced in Java 5 and strategically aligns itself with the current trend towards a more document–centric messaging model. The use of annotations simplifies the implementation and eases the service development. Based on Plain Old Java Objects, containing the implementation of the WS interface, the annotations are included in the code describing details such as service identification, SOAP binding, namespace and operation descriptions, among others. All these details are instantiated during the merging step of the code generation and they are used to be compiled into byte–code ensuring better platform independence for Java applications.

JAX-WS uses JAXB as default data binding to process the message marshalling/unmarshalling. These operations map the Java types into WSDL types and vice versa. The resulting mapping called WS *method stubs* are part of the final service and they are used to communicate with the client all along the invocation. In terms of solutions supporting JAX-WS, Sun Metro[5] implements all the specification and it is distributed on major application servers. Metro needs to be configured on the basis of Apache Tomcat server,[6] the stack engine to publish services and supports additional needs like MTOM, useful for the service attachments manipulation.

The services have to fulfill a format and configuration for deployment. Following the Tomcat server architecture, all services are deployed in form of a Web Application Archive (WAR), a special JAR file used to distribute a standard Web application. In the case of the *jigsaw* framework this archive includes configuration files (`sun-jaxws.xml` and `web.xml`); the description of the resource; the wrapped CLI application; the dependencies, when they are necessary; and the Java classes representing the WS interface and the stubs.

During deployment the services container sets up new services at runtime without interrupting its normal operation (i.e., there is no need to restart the server). This quality, known as *hot deployment*, is a trending feature implemented in current technologies such as OSGi[7] or Apache Tomcat. A service is identified by two elements that are unique in the deployment scope. First, the service name, a combination of the symbolic name and the service version assigned during the characterization of the description. Second, a service location, that is interpreted as a URL pointing to the description (WSDL) of the Web service. Just after the deployment, this description is available becoming the service contract. This service is ready for invocation. Removing the deployed services releases safely the reference to the service from the container; and from the list of services published on the server.

[3] Apache Axis: http://axis.apache.org/
[4] JAX-WS: http://jax-ws.java.net/
[5] Sun Metro: http://metro.dev.java.net/
[6] Apache Tomcat: http://tomcat.apache.org/
[7] OSGi: http://www.osgi.org/

3.1.2 Runtime dynamics

The core functionality of the *jigsaw* framework, besides the provision of services, is the instrumentation of the logic related to the interpretation of the arguments, dependencies configuration, and the execution. This operative process is organized in three parts: data marshaling, resource allocation, and data management.

Data marshaling

A service has arguments described with different data types and structures. Similarly, the results of an execution should match the description of provided outputs. The framework takes the original results of an execution and forwards them preserving that structure and data typing declared on the description of the CLI application. This interpretation is mandatory to correctly process inputs and outputs of the service. Although the description of a Web service provides the basic information of arguments, this information is not sufficient to identify implicit CLI parameters nor the special connotation of a unique reference (i.e., URI address) as a group of multiple files representing a specific format. Furthermore, when the results are not files, they are commonly presented in the standard output as sequences of strings. This is the reason to interpret these outputs after execution using the description of the application. This task involves parsing, casting, and mapping the results into the right structure to finally return the expected value. The framework reproduces as much as possible the structure organization resulting from the execution.

In the *jigsaw* framework the resulting structures are defined as (nested) arrays when the output represent more than a simple value. Nevertheless the interpretation of outputs is not trivial because each application may represent its results organization arbitrarily. If an application provides such a result, the output should be adapted for being *jigsaw*–compatible. This is done using the nesting properties defined in the description.

In practice, the interpretation of parameters is done using data marshaling/unmarshaling in two different circumstances. First, the transformation between Java native types and SOAP messages to communicate during service invocations is performed by the servlet engine using JAXB. All parameters of the description are represented in Java source code and then compiled in order to be interpreted by the Java Virtual Machine. This transformation is exhibited in the WSDL document and the schemes of the service messages. Second, internally the framework also performs other transformations to interpret the description of the application at runtime associating the correct types and structures of the incoming input data, and matching the results. This dynamic transformation associates non–typed data into Java objects before and after the dynamic resource allocation.

Resource allocation

The execution instrumentation interprets the description of the application building the complete command–line to execute, resolving the inputs, and setting up the environment of execution. On the execution endpoint, an isolated sandbox is created and then all necessary data is retrieved on it

before execution. This execution can be performed directly on the services container host as a *local* execution or it can be strategically delegated using distributed computing infrastructures. The local execution is the simplest and the default *jigsaw* runtime instrumentation. Since the application runs in the same place where the service is hosted, multiple instances of heavy–demanding applications are not suitable and a remote relocation should be contemplated for these cases.

Following the proposed hybrid model introduced in Chapter 2, a remote execution of an application may be performed transparently but it requires relocation of the resources on remote endpoints and additional management of the infrastructure components. On the EGI production grid, for example, this execution implies using several components of the grid infrastructure such as the Workload Management System (WMS), file Storage Elements (SE), the Logging and Bookkeeping (LB) service, and Computer Elements (CE). The correct execution on the grid involves strategies from the submission to monitoring procedure or alternative mechanisms of execution like pilot jobs [Casajus et al., 2010]. A sequence diagram of the steps during grid execution is presented in Figure 3.5. The diagram shows a simplified sequence of the *jigsaw* operation invocation:

1. The client invokes the execution operation of the Web service.
2. The application is submitted to the grid using a WMS.
3. The actual execution is delegated to a CE and a job identifier is registered on the LB service.
4. The Web service, acting as submitter, receives a job identifier to trace the progress.
5. The input data is staged from a source. This source may be a database, and SE, etc.
6. The command line is built with the fetched data.
7. The CLI application is fired.
8. The Web service ask the status of the execution periodically (loop).
9. The LB service check the status of execution on the CE (loop).
10. The CE provides the updated status of the execution (loop).
11. The status is returned to the Web service until it is done (loop).
12. After a successful execution the results are saved on a SE.
13. The Web service obtains the references of the results.
14. The references to those results are returned to the Web service.
15. The client receives the results of the execution.

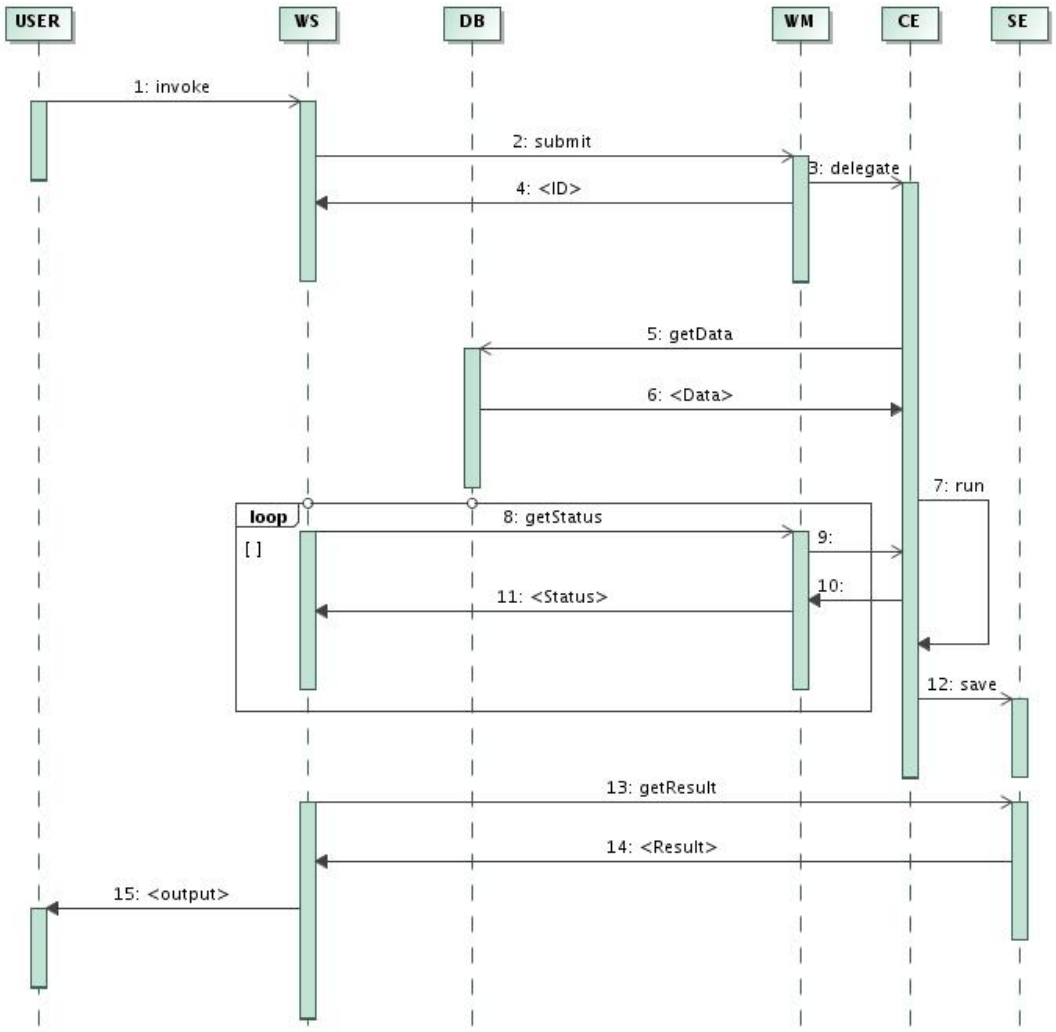


Figure 3.5: Execution sequence on production grids

Data management

Part of the instrumentation process involves the data transfer resulting from the file manipulation during execution of services. The data management is necessary to provide inputs to services, store results and return them to the client. Indeed, for performance reasons, the files themselves are never transferred as part of the service invocation messages; a dedicated data transfer mechanism is used instead. Furthermore, on distributed infrastructures, files are directly transferred between nodes and they never transit through the *jigsaw* engine which would become a potential bottleneck in data–intensive applications. On the other hand, if the service receives references to files, they are staged to the execution place managing the different protocols such as grid FTP or HTTP, and file schemes like the grid LFN or local FILES.

Two scenarios are figured out regarding data transfer after the execution of an application. In the first scenario, *jigsaw* publishes into a public space all the resulting files if they are inaccessible to the remote client. Usually this case happens in a local execution when the outputs are defined using local file references. For this case, a translation of the reference location is performed in

favor of a suitable protocol, by means of data transfer between the execution point and the final storage place. In the second scenario, *jigsaw* registers all resulting file produced during the execution on a remote storage resource, then reports the file references to the client. In both cases files are delivered to the client using additional data transfer operations. This scenarios guarantees the permanent availability of results as well as scalability. In fact, scalability is never affected because data is processed during the metadata resolution without explicit data transfers. This implies data is never associated to the application, and the final results stay persistent contrary to the execution instance.

3.1.3 Invocation of services

The generic client API to invoke services is the third module of the *jigsaw* framework. The API provides methods to interpret a WS description, and invoke an operation with the input arguments. Besides, it is possible to use the same API to invoke third–party Web services as long as such services meet the JAX-WS specification, and the declared data does not define personalized messages.

The API defines parsers for interpreting the WSDL and their associated schemes; and consumers for dispatching the messages to the server and obtaining the results. The parsers define several signatures of the following methods:

- `getServices`, to obtain the endpoints on the service,
- `getPorts`, to obtain the implemented ports of the endpoint,
- `getOperations`, to obtain the declared operations of a port,
- `getRequestSequence`, to obtain the elements expected by an operation, and
- `getResponseSequence`, to obtain the elements expected as the result of an execution.

The consumer may reuse the information provided by the parsers to consume a WS operation. It also defines several signatures of the following methods:

- `dispatch`, to invoke an operation synchronously,
- `invoke`, to submit an input request asynchronously, and
- `getResponse`, to obtain the results from an asynchronous dispatch.

Using Web services the interoperability is granted between clients and servers thanks to the messaging protocol independence. Consumers dispatch a well–defined message and wait for the result. This action is possible creating messages with the references retrieved from the description associated to their corresponding values and send them to the server. The *jigsaw* API client implements a dynamic method to consume Web services. This method involves a generic dispatch client that offers flexibility to reuse the same operations to perform the marshaling/unmarshalling and invoke different Web services. The dynamic method is a pure XML messaging–oriented client and requires advanced use of SOAP message construction and interpretation because each operation

provides a different response message. Despite the different SOAP messages, since each server implements the same definitions with different formats, the processing of the server response is performed transforming the SOAP messages into objects that can be interpreted by the data binding of the framework. Nevertheless, it is necessary to pay special attention to the format specificness of the SOAP messages because in practice is not possible to test all types of SOAP message implementations. In the case of *jigsaw* the support of Metro messages is granted to parse and execute the services based on the JAX-WS specification. Description of services can be interpreted for other types of services but the dispatch and processing of incoming messages is not possible due to potential implementation incompatibilities.

3.2 Non–functional concerns integration

Non–functional concerns define the expected qualities of a system that are not associated directly to the business logic of the framework. They are constraints, requirements or goals observable in parallel to the normal behavior of the system. Several non–functional concerns can be integrated within the *jigsaw* framework. For example, we developed support for three non–functional concerns to address the needs of the NeuroLOG and VIP projects: (i) a strong and distributed access control policy to prevent unauthorized invocations, including logging and accounting, (ii) semantic annotations support, and (iii) multi–platforms execution.

3.2.1 Access control, logging and accounting

The support of access control, logging and accounting are optional in the framework like all non-functional concerns. However access control plays a different role compared to logging and accounting. Access control is a major concern for authentication and authorization that must be enforced permanently in distributed environments [Gaignard and Montagnat, 2009]. It is a low–level architecture layer based on the management of user credentials validated by external certification authorities [RFC 5280]. Therefore, access control must be implemented within a system environment accordingly to reference standards (e.g, X.509 for public key infrastructure or the TLS/SSL secured transport layer). Conversely, the logging and accounting are only integrated for monitoring the execution of services though an ad–hoc implementation. They are not used to accomplish the executions however these concerns are required in the context of the system deployment. Logging and accounting provide a mean to track services across the framework during the complete lifecycle but their introduction into the framework or their absence is up to the manager.

These concerns can be integrated within the *jigsaw* framework without impacting the data modeling nor the core application. The access control introduction involves to manage user's credentials before the execution of the CLI application. The credentials are passed to the service provider as part of the headers of the SOAP message to invoke transparently an operation. Thus the execution processing is not modified with the inclusion of credentials validation. On the other hand, a similar code integration to access control, logging and accounting is reflected during the service generation. The source code of both kind of concerns can be inserted into the template in order to be merged with a description instance. This process generates the final Java code used to create the

WS interface, as is shown in Figure 3.6. Finally, each concern is automatically enabled at runtime once the libraries implementing these requirements are added to the framework dependencies.

```
1  package jigsaw.ws;
2  ⋮
3    @Resource private WebServiceContext wsContext;
4
5    private boolean checkAuthentication() throws ServerException {
6      AuthorizationManager authorizationManager = null;
7      try {
8        authorizationManager = AuthorizationManager.getInstance();
9      } catch (IllegalAccessException e) {
10       throw new ServerException("User authentication failed");
11     }
12     return authorizationManager.isAuthorized(wsContext, "exec",
13       "$application.getSymbolicName()-$application.version");
14   }
15
16   @WebMethod(operationName = "local")
17   @WebResult(name = "localResult", targetNamespace = "http://i3s.cnrs.fr/jigsaw")
18   ⋮
19       JigsawOutput$suffixclassname output = new JigsawOutput$suffixclassname();
20       try {
21         if (checkAuthentication()) {
22           AuthorizationManager authorizationManager = AuthorizationManager.getInstance();
23           String callerDN = authorizationManager.retrieveCallerDN(wsc);
24           TraceManager.getInstance().genTrace(callerDN, "Invocation of service : " +
25             "$application.getSymbolicName()-$application.version");
26
27           Object[] objects = null;
28           Description description = DescriptionFactory.getInstance(this);
29           ⋮
30 #else objects});
31 #end
32         } else {
33           TraceManager.getInstance().genTrace(
34             AuthorizationManager.getInstance().retrieveCallerDN(wsc),
35             "Unauthorized invocation of service : " +
36             "$application.getSymbolicName()-$application.version");
37           throw new ServerException("Your are not authorized to invoke this tool");
38         }
39 ⋮
```

Figure 3.6: Snapshot of the modified template listed in Figure 3.3 including in bold the code snippets for access control (`checkAuthentication` operation) and accounting management (`TraceManager`)

3.2.2 Semantic annotations

Semantic representation of information has become broadly used to enhance platforms with domain–specific knowledge. This representation aims at facilitating platform usage, sharing of experimental data and results, and experiments themselves fostering collaborations. Ontologies, in

domain knowledge conceptualization, became a cornerstone for the underlying information systems, as they are built upon controlled vocabularies, logical constrains and inference rules. SOA, generally relates to those platforms, provides dedicated tools for the publication, the identification, and the invocation of services. However the technical description of services, like WSDLs and data schemes, does not provide any understanding on the nature of the information processed nor on the applied operations. Therefore, the exploitation of catalogs of data processing services requires a clear understanding of how data is processed and the nature of the data transformation implemented by the services.

Generating semantic annotations, before execution to validate inputs, during processing or even after execution to add the new information to the knowledge base, implies matching technological concepts like elements of the service messages with concepts specific to the application domain that represents high level characteristics. This generation may reuse the WSDL information and the intrinsic information of the CLI application contained in the *jigsaw* description published as part of the Web service. The use of the CLI description tends to explicit the understanding of the nature of processed data, and the nature of the information of the applied processing to benefit, both at experiment design–time and runtime. This approach tackles three aspects of semantic services, leveraging existing ontologies to describe generic information as well as domain–specific nature of data and processing tools [Batrancourt et al., 2010]: (1) it clarifies the binding between service descriptions and domain concepts through a taxonomy; (2) it enables the coherency of service composition design; and (3) it makes possible to infer new knowledge along the platform exploitation. This last point is achieved by describing reusable domain–specific knowledge inference rules associated to specific natures of processing. The application of these rules on a semantic database containing traces of services invocations enriches the platform with new valuable expert information.

In the context of the VIP project, the semantic annotation of *jigsaw* services were integrated through a dedicated user interface of the client application, while the record of provenance information is stored at runtime by the service invoker. Conversely, queries of the provenance information enable to retrieve all available annotations in order to define explicitly the semantics of its models and simulations. The integration also provides the formal description of those applications to be referenced semantically.

3.2.3 Multiple infrastructures execution

To support multiple infrastructures, the model described in Chapter 2 distinguishes the application description from the implementation(s). Each implementation may be defined for several platforms. At the same time, a platform includes execution profiles, and holds information about the artifacts and the target application. In addition, the customizable execution environments may be defined for a specific profile or shared among the platforms of the declared application releases.

At programming level the *jigsaw* framework defines a general interface to implement the execution binding for each computing infrastructure such as the default local execution provided by its core module. This represents an intermediate layer for a developer and the core framework. The developer is interested in creating a connexion with a new infrastructure. Complementary,

the framework processes the invocation parameters of the application (before performing the submission) and provides the results after the execution. This interface is defined as a set of procedures that should be overwritten to implement: access to the target infrastructure, job instance submission, and application execution monitoring. The resulting execution strategy also involves to modify the service template to define an operation representing the invocation to the infrastructure. Both, the template and the implementation code, are then included into the framework as an additional library for the generation of the service and its execution. Following this approach other bindings like GASW, as part of the VIP platform [Ferreira da Silva et al., 2011], were successfully integrated in the framework. In this case, VIP reuses transparently the *jigsaw* data binding, and the user's interfaces to build services while the generation of the business code is overridden along with the implementation of the CLI applications executor. Thanks to this extension, users not only can dispatch executions on multiple infrastructures like PBS clusters or production environments like the European Grid Infrastructure (EGI), but they also can continue to process the input arguments and output results with *jigsaw*. The instrumentation of new execution strategies does not require extensive development because the *jigsaw* design abstracts the notion of independent execution strategies without affecting the rest of the framework organization. However, the implementation of a new strategy is not very common once a suitable execution method of a given platform is defined.

3.3 Framework integration into third–party software

The *jigsaw* framework was designed to cover stringent data flow manipulation capabilities as those enacted by a demanding scientific workflow engine. In fact, the adoption of a standard WS interfaces make *jigsaw* completely independent from any platform. It can be used with any WS–compliant engine (e.g., Taverna [Oinn et al., 2004], Triana [Taylor et al., 2005], BPEL [WS–BPEL, 2007]) or even standalone applications through a generic WS client. In this perspective the *jigsaw* framework has been integrated into third–party software at several levels: development API, and comprehensive integration. As development API the integration reuses exclusively the provided operations of the framework. The comprehensive integration, on the other hand, assemblies several projects to provide a end–to–end framework to users.

3.3.1 Development API

The modular conception of the framework allows developers to reuse the graphical interface, the set of libraries implementing the runtime dynamics, or the generic invocation client as independent modules in their own software. As a matter of fact, *jigsaw* is integrated in this way in the NeuroLOG middleware. The components for the generation of services and execution are embedded into the client interface, and the libraries of the *jigsaw* framework are configured on the server side for the correct execution of CLI applications and processing of results. The framework also handles the sensitive data used as input of those applications through the data management module. Finally, non–functional concerns are integrated in accordance with its requirements of

non–centralized and secured platform by a personalization of the template that is used during generation of WS interfaces.

3.3.2 Comprehensive integration

Clients such as the MOTEUR workflow enactor [Glatard et al., 2008] interface with the application tools through the *jigsaw* client API that facilitates the WS interface parsing and invocation. In addition, new execution strategies are included like external concerns in larger frameworks. Through *jigsaw*, a client application is shielded both from details of the CLI tools invocation and from the grid invocation interface, including data handling and Grid security credentials management. Its role stays focused on analyzing the data flow and enforcing the coherent execution of the application in a distributed environment by delegation to the *jigsaw* system. Specifically, an end–to–end framework that facilitates the *gridification* of applications and their executions on different DCIs has been implemented as a natural follow–up of the implementation in combination with other relevant projects, namely VL-e Toolkit, MyProxy, and DIRAC.

The overall framework architecture is depicted in Figure 3.7. The MOTEUR client is the front–end component that connects the user to the rest of the framework. It is also the working environment where users manipulate their applications at design time. A MOTEUR client interacts with the MOTEUR server at runtime to execute the applications with a specific dataset. The MOTEUR server is responsible for invoking each application deployed locally or remotely through generic interfaces. The final CLI application, encapsulated by *jigsaw*, is submitted to a DCI by means of an intermediate middleware such as DIRAC. During the execution, a user's credentials may be needed for authentication with the infrastructure, thus all of framework components can connect to a MyProxy server to fetch a proxy certificate. Finally, data transfers between executions are enabled through the VL-e Toolkit[Olabarriaga et al., 2010] because it provides a unified view of heterogeneous file systems.

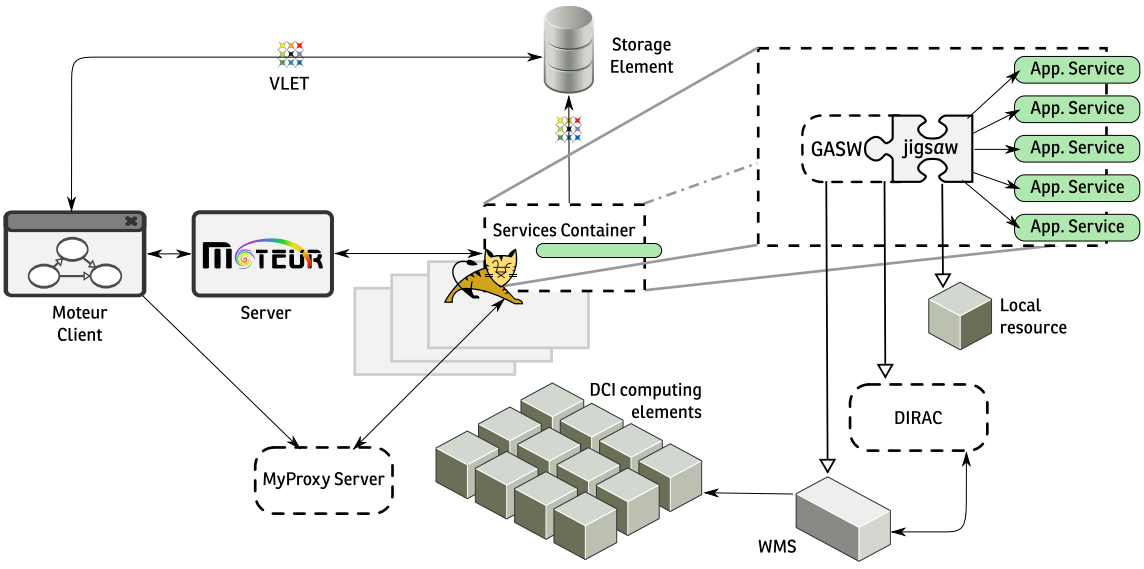


Figure 3.7: Architecture of the integration of *jigsaw* in a comprehensive framework

MOTEUR, targets a coherent integration of a data–driven approach to achieve transparent parallelism and manipulate complex data structures. The MOTEUR client provides to users a graphical interface to configure services and describe the semantics of data flows. The description is represented by the GWENDIA workflow language [Montagnat et al., 2009], that supports the required expressiveness to represent services composition. While the MOTEUR client offers design tools to build workflows and configure an environment of execution, the MOTEUR server provides asynchronous invocation and orchestration of services, and improves the execution of large–scale, data–intensive workflows. The integration of *jigsaw* with MOTEUR provides to final users a full range of functionality, facilitating the reuse of scientific applications, their composition, and large-scale experimentation.

Addressing the credentials management task, the framework supports two alternatives to create and renew proxy certificates. The first one is to create a proxy directly from the user's certificate file and private key granted by any Certification Authority (CA). The proxy may be used in the MOTEUR client where the user's credentials are available. Since at runtime services are invoked remotely from the MOTEUR server or a Web services container, a second alternative is used to download a proxy from any MyProxy server [Kouril and Basney, 2005]. The user is just required to provide the login and password of the credential stored on the MyProxy server. The validity of the proxy is checked each time a connection is performed. An expired proxy will be automatically renewed without interrupting the entirely execution of the application.

The MOTEUR client uses the operations implemented by the VL-e Toolkit to download services descriptions and upload artifacts required for service execution. It is also used for file staging on the services container to provide the data inputs to the application.

In summary, the integration of each tool is effective at several levels. At the front–end level, this integration provides an interface to *gridify* scientific applications, and to invoke the deployed services by means of standard Web services mechanisms or convenient bindings accessing the distributed infrastructure directly. At the back–end level, the integration gives a transparent access to multiple DCIs. It brings to the user the ability to execute the applications on those DCIs with the same execution and enactment engine.

3.4 Implementation outcomes

In spite of the reference implementation, some pitfalls are not resolved consistently. The jobs submission API delivered by EGI is not mature enough to be used directly in the framework in the submission of jobs or in the implementation of fault tolerance mechanisms. For this reason, the execution on distributed infrastructures is not natively introduced because there is no interface available to work directly at the programming language level. It means that external bindings to connect the execution manager with the framework are implemented overriding the default submission mechanism as an external execution strategy using third–party tools like the DIRAC pilot framework and workload manager system.

At the data model level some arrangements fix the serialization of file references, because the serializer of the server does not resolve the use of URLs as expected. In fact, the implementation of

the framework represents a challenging endeavor facing defects of external components that are out of the scope of the implementation but should be managed and consolidated.

The implementation of submission strategies on other distributed computing infrastructures are potential extensions of the framework in future milestones. Nevertheless, this implementation effort is barely profitable where the relocation of resources is not exclusively yielded to *jigsaw* and it depends on the configuration of additional technological layers. For example, the DIET toolbox,[8] requires the deployment of several agents for operation, so the management of such elements represents more than an external concern directly associated to the execution of the CLI application.

The long–term goal of *jigsaw* is to provide an automatic execution mode where users without technical skills could find difficult the selection of execution strategies (local or remote) because they do not always have a clear idea about the implications of the application execution on distributed computing infrastructures. Moreover, users do not know the load endured by the server nor the status of such infrastructures. These arguments show the necessity to provide an strategy to choose automatically the type of execution on behalf of the user. The explicit operations still remain relevant though.

3.5 Conclusion

The reference implementation framework presented in this chapter is not only a proof of concept showing the feasibility and the relevance of the model merging the SOA principles and the global computing implementations in combination with enhanced CLI application descriptions. This framework meets real requirements of users facing complex issues to resolve their needs of reuse, fast and reliable execution. *jigsaw* is the result of a requirements analysis trying to guarantee the use of compelling CLI applications, embracing at the same time lead technological evolutions such as SOA and the distributed infrastructures. The proposed framework is a flexible solution to execute legacy tools while delivering the following features:

- complete lifecycle of services providing a manageable work environment,
- transparent execution resulting from dynamic resource allocation,
- remote execution by direct invocation or delegation,
- compliance with standard protocols during message transactions,
- data staging for execution and results processing,
- comprehensive management of I/O arguments,
- awareness of application dependencies,
- integration of non–functional concerns, and
- reuse of components in third–party infrastructures.

[8] DIET toolbox: http://graal.ens-lyon.fr/~diet/

The *jigsaw* framework was implemented to address specific requirements of the NeuroLOG project. Nevertheless it is used in various external projects such as GWENDIA or VIP exhibiting their generic approach. This approach also enables compositions of scientific workflows using MOTEUR with strong type mapping and complex structures as described in the second part of this document. This solution is a step forward the bridge of CLI applications with modern service–oriented architectures providing a clean and simple set of tools to assist scientists that are not computer specialists to build, run, combine, and share their work.

Part II

Scientific Workflows in Neuroimaging

Chapter 4

Scientific Workflows

Nowadays, the reuse of software components has an important impact in e–Science [Pagni et al., 2008; Geddes et al., 2005]. Thanks to Web standards, applications can run in distributed locations exchanging information and being combined more readily than ever before. Web services have been successfully used in the scientific domains such as bioinformatics and medical image processing [Labarga et al., 2007; Glatard et al., 2006b]. In fact the need of interoperability and increasing demand of computing power enforced the implementation of frameworks to assist in the reuse and distributed accessing of services as shown in Chapter 3.

Users also often describe and enact their applications by orchestrating multiple services into pipelines. This process involves choosing a set of appropriate services based mainly on functional properties, to arrange them in sequence according to the application logic by solving the connectivity between services, and to convert the complex process into a target workflow language which can be executed on a computing platform.

Scientific processing pipelines are often composed of many applications dealing with large datasets running in specific environments. Among the involved applications, some cannot be executed before the termination of its precedences due to control or data dependencies. On the other hand, several applications are independent which means they can be executed in parallel. These features impose to take advantage of parallelism, and execution interoperability.

This chapter presents the salient features of scientific workflows. It also introduces a workflow definition language, and a workflow enactor engine suitable for the efficient composition of services and parallelism exploitation in the perspective of software reusability.

4.1 Elements of scientific workflows

A scientific workflow (aka data intensive workflow) is an orchestration of coarse–grained processes [Bharathi et al., 2008]. Scientific workflows are designed to support the automation of complex, service–based and data–intensive applications. They combine a dataflow model, whereby a workflow consists of a set nodes (activities) that are connected through data dependencies links, with a functional model that accounts for collection–oriented processing [Missier et al., 2010]. This combination of models is designed to strike a balance between expressively and simplicity.

Scientific workflows, as data–driven languages, separate explicitly the definition of data to process from the processing logic. This separation is convenient because the same workflow can be reused with different datasets without any change. This separation is commonly observed because applications are made available independently to the data to process. The data driven approach is also appealing to the Grid community because of implicit parallelism. Indeed, a workflow application graph expresses parallel enactment, and the data parallelism is expressed through the multiple input datasets pushed into the workflow.

Several abstractions were introduced to express the data representation in scientific workflows. For instance, the Swiftscript language [Zhao et al., 2007] defines *arrays*—indexed collections of data items with homogeneous type, as first–class entities. In an analogous way, the Simple Conceptual Unified Flow Language[1] (SCUFL) used in the Taverna workflow management system [Oinn et al., 2006b] refers to list of data items to represent indexed and typed collections of elements. This latter definition corresponds to an equivalent concept of array. The use of arrays represents a practical way to exploit data parallelism based on array programming principles.

The introduction of array programming concepts eases the description of mathematical processes involving arrays [Hellerman, 1964]. Array programming aims at simplifying the manipulation of data structures at formal level. In array programming, originally an array is considered as first–class entity. It is thus used directly with traditional operations like the addition. These operations are defined natively to operate on arrays or on combinations of scalar values and arrays. For instance, $X + Y$ and $k \times Z$ are valid expressions operating on each array element, where X, Y, and Z denotes arrays and k any numerical value. Therefore, array operations are a convenient way to explicitly working without loops for iterating operations over collections. In fact, they reduce the use of control structures inside the formalization.

Operations on arrays may be extended to object–oriented languages [Mougin and Ducasse, 2005]. This extension introduces the application of methods on arrays of object, and/or the application of methods to arrays of parameters. Array may be expanded into its elements and each element is treated as an individual item in further processing. Hence, this processing applies to individual elements of the expanded array. In the same way, other array operators may also be defined such as *reduction* of an array into a resulting scalar; *compression* as form of evaluation of a test over all elements of the array; *transposition* to rearrange the array elements or re–size it; *join* to search for indexes of selected elements; or *sorting* to obtain an array in a defined order [Montagnat et al., 2009].

Arrays may be nested at any depth defining new data types of array of objects. Any given data item is therefore always associated with a type, and its corresponding (multi–dimensional) integer index with a dimension per nesting level. For example, the array $W = [[[a,b]],[[c]],[[]]]$ is a 3–nested levels array of characters and w_{001} designates the character b. It is possible to represent elements of arrays with the special value $\varnothing$ as the absence of data. This value is particularly important to represent placeholders in an array preserving indexes.

The data–driven definition of scientific workflows through high level interfaces empowers users, who may have limited understanding of programming, to assemble advanced applications pipelines

[1] SCUFL language: http://www.mygrid.org.uk/usermanual1.7/scufl_language_wb_features.html

involving complex data structures. Although, scientific workflow enactors are in principle similar to traditional programming environments. They are based on a language specification that is then interpreted. Nevertheless, traditional environments do not deal with parallelism and dataflow aspects, and they are not oriented to work with large scale of data or a high computation abstraction. In this section is presented a representation of the dataflow during the enactment of workflows, as well as the elements that define their structure.

4.1.1 Activities and dependencies

A workflow activity is an atomic process that is bound to an arbitrary number of input and output ports. The ports represent data buffers where data items to process are received or produced data items are stored after firing an activity. Input and output ports are typed. The output port types define the activity type. The activities have input/output ports with a defined *depth*. The depth of a port determines the number of nesting levels the input port will collect or the output port will produce. It impacts the number of firings of the activity considered. Activities may receive inputs with different nesting levels. The usual behavior of an activity receiving a nested array is to fire once for each scalar value embedded in the nested structure. However, there are cases where the semantics of the activity is to process a complete array as a single item rather than each scalar value individually. An important property of activities invocation in an asynchronous execution is that multiple invocations of an activity on array items preserve the array indexing scheme.

The dependencies between activities are defined with links. Data links interconnect one activity output port with one activity input port defining data dependency between two activities. In some cases, there is no data dependency explicitly but an execution order should be preserved. A control link interconnecting processors may then be defined.

Scientific workflows are composed by many activities with inter–dependencies which define ordering constraints at execution time. Activities may be instrumented as services processing data at programming level (i.e., Java beanshells or R scripts) or invoking operations of standard implementations such as Web services. Each workflow data link is associated to a service argument. For instance, a WS message may represent the collection of input ports of an activity, where the data structure and types are defined in the WS description. The expressiveness of the activities composition depends on the availability description of services, thus the more complete is the description of involved applications, the more precise is the service composition. The emphasis put on the description model of applications in Chapter 2, and the implementation as standard Web services described in Chapter 3 represent the effort of this work to provide the suitable information of services for their composition.

4.1.2 Iteration strategies and control structures

The concept of *iteration strategies* [Oinn et al., 2004; Sroka et al., 2009] defines the combination mechanism for input data items received on several input ports of a same activity. They define the number of activities fires and the input data sequence for each invocation. Iteration strategies are also responsible for defining an indexing scheme that describes the items from multiple input

nested array resulting in an nested output array. Iteration strategies were first introduced in the SCUFL language to combine complete iteration expression trees. Hence they produce complex iteration patterns without requiring to define any explicit loop.

Data parallelism is completely hidden through the use of arrays. Advanced data composition operators are available through activity *port depth* definitions, representing the dimension of the array, and iteration strategies. Complex data parallelisation patterns and data synchronization can therefore be expressed without additional control structures. Only conditionals and loops expressions are needed to control the dataflow across the workflow. Conditionals represent an array–compliant *if–then–else* kind of structure. Alike, a loop represents a *while* or a *for* kind of control structure. The syntax associated to these structures is detailed in [Montagnat et al., 2009].

4.2 GWENDIA & MOTEUR

Among the existing scientific workflow environments (e.g., Taverna,[2] Triana,[3] Pegasus,[4] Kepler[5]) the use of the GWENDIA language on MOTEUR targets a coherent integration of a data–driven approach to achieve transparent parallelism in a comprehensive framework by manipulating arrays. GWENDIA conditionals and loop control structures provides the required expressiveness to represent services composition and data manipulation; and the asynchronous invocation of services of MOTEUR optimizes executions on a distributed infrastructure. These characteristics make GWENDIA and MOTEUR suitable for the purpose of reuse of software components and interoperability.

4.2.1 GWENDIA: a workflow definition language

The Grid Workflow Efficient Enactment for Data Intensive Applications specification (GWENDIA) [Montagnat et al., 2009] is a data–driven language for the description of complex application dataflows. It targets the coherent integration of array manipulation, control structures, and efficient asynchronous representation for execution of workflows. GWENDIA defines data types, processors, ports, links, iteration strategies, and control structures in a compact XML format inspired by SCUFL. The syntax of the most relevant elements are detailed below.

Processor

A processor is a workflow activity representing a service. Several types of processors are defined: bean shells, Web services, etc. Special cases of processors are: **source**, without inbound connectivity delivering external data values as inputs; **sink**, without outbound connectivity receiving final workflow results as outputs; and **constant**, delivering a single constant value.

[2] Taverna workflow management system: http://www.taverna.org.uk/
[3] Triana project: http://www.trianacode.org/
[4] Pegasus workflow management system: http://pegasus.isi.edu/
[5] Kepler project: https://kepler-project.org/

Port

Ports are the inputs and outputs of processors. Ports are identified with a name, type and depth. Since data manipulated in the language is typed, four basic types are defined: *integer*, *double*, *string*, and *file* (i.e., URIs referencing files). Scalar items have depth equal to zero. Data with homogeneous types may be grouped in arrays.

Iteration strategies

Four types of iteration strategies are defined:

1. The *dot product* $\{\odot\}$ matches data items with exactly the same index in an arbitrary number of input ports. The activity fires once for each common index, and produces an output indexed with the same index.

2. The *cross product* $\{\otimes\}$ matches all possible data items combinations in an arbitrary number of input ports. The activity fires once for each possible combination and produces an output indexed such that all indexes of all inputs are concatenated into a multi–dimensional array.

3. The *flat cross product* $\{\ominus\}$ matches inputs identically to a regular cross product with a difference in the indexing scheme of the data items produced. It is computed as a unique index value by flattering the nesting–array structure of the regular cross product.

4. The *match product* $\{\oplus\}$ matches data items carrying one or more identical user–defined tags, independently of their indexing scheme. Its output is indexed in a multiple nesting levels array which index is the concatenation of the input indexes.

Conditionals

A conditional has an arbitrary name of inputs, a test expression to evaluate for each data received from the input ports, and an arbitrary number of paired outputs corresponding to the *then* and *else* branches.

Loops

Two types of loops are defined:

1. The *while* kind of structure. It is composed by an expression used to stop the evaluation; input ports receiving the loop initialization value, and the values that loop back tho the activity after the iteration; and the output ports receiving a value when the condition become false, and all values from either the initialization or the looping part.

2. The *for* kind of iteration structure. It has the same elements of the other loop but the number of iterations is the same for all initialization value.

Filters

Filters are particular manipulation activities that modify nested array structures. It is useful to discard results that have not passed a condition, whereas the indexing of resulting items does not need to be preserved, or to combine content of two complementary arrays with the same structure. Examples of filters are the *split*, *merge*, and *concat* operations.

Links

Links are simple data dependency declaration connecting input and output ports.

4.2.2 MOTEUR: a workflow enactor

MOTEUR [Glatard et al., 2008] is an enactor engine designed for executing workflows consisting of standard services or customizable processors embarking user–defined source code. MOTEUR exploits service parallelism at workflow level and data parallelism for multiple datasets. It responds to requirements of a suitable scientific workflow environment [Maheshwari, 2011]:

- **Scalability and optimization**. The performance remains constant albeit the number of tasks without considering issues related to a distributed infrastructure such as network latency or protocol offsets. In terms of optimization, MOTEUR performs service grouping leading significant speed–ups, especially on infrastructures that introduce high overheads.
- **Data description and management**. Data is described in a grammar allowing types and multi–dimensional arrays. MOTEUR only uses references to files providing implicitly access to shared data repositories. The use of references simplifies the data management avoiding potential bottlenecks of data staging. Additionally MOTEUR may trace the provenance of data.
- **Interface to Distributed Computing Infrastructures**. Seamless access to remote infrastructures is granted by the service processing model. MOTEUR acts as a client executing applications on production grids (e.g., EGI), research clusters (e.g., Aladdin/Grid'5000) [Montagnat et al., 2010], and it is also interfaced with the HiPerNET cloud [Truong Huu et al., 2011].
- **Expressiveness**. A rich semantic of the workflow specifications simplifies handling of activities and data. It represents the interface between the enactor an the user as a transformation language. MOTEUR implements control mechanisms and iteration strategies defined in GWENDIA.
- **Usability**. It enables an easy and convenient composition of workflows through the graphical interface. MOTEUR is compatible with workflows written for Taverna. Finally, the architecture of the enactor facilitates the extensibility of the environment by implementing new types of activities or linking to new application types.

MOTEUR is also a graphical environment for the design of scientific workflows. The user interface provides all the elements to create, enact, and trace the provenance of data. Figure 4.1 shows a screenshot of the environment.

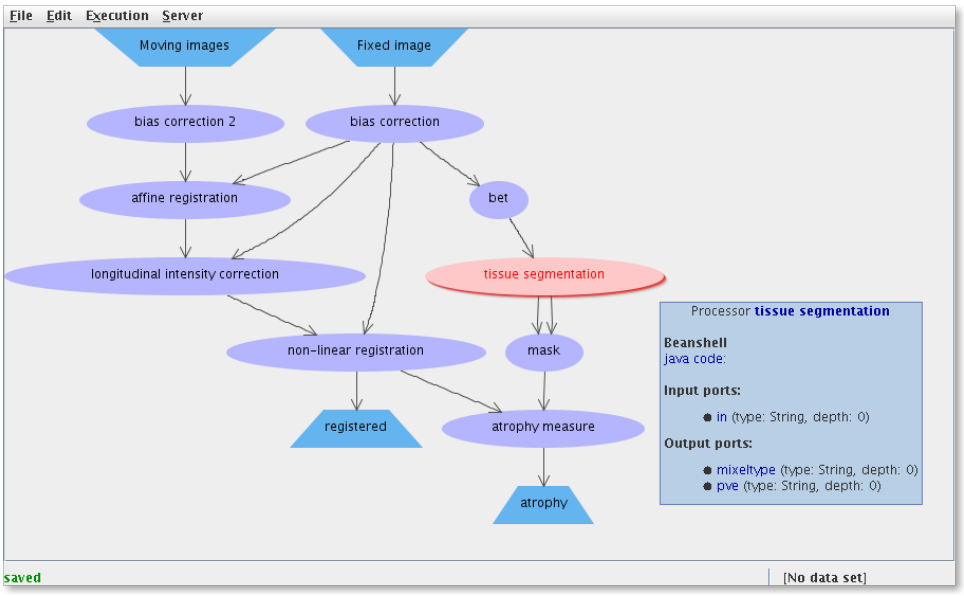


Figure 4.1: Graphical interface of MOTEUR

4.3 Summary

In this chapter, scientific workflows were presented as an expressive and efficient approach to describe complex data processing based on reusable modules. As a result of this study, the separation of data and processing logic in combination with the parallelism exploitation are identified as salient elements to grant an easy enactment of data–intensive applications on service–oriented architectures. MOTEUR and GWENDIA were also selected as a promising implementation of scientific workflows because they respond to the requirements of a suitable workflow environment. Both initiatives adopt an advanced representation of data using array programming and they integrate transparently the use of distributed computing infrastructures.

Starting from the wrapping of CLI applications as services detailed in the fist part of this document, in the following chapters scientific workflows are used in the service composition of neuroimaging analysis use–cases. In those experiments the focus is put on the expressiveness and the grid computing exploitation to obtain time execution improvements and to contribute in qualitative analysis of neuroscience studies. ◊

Chapter 5

Neuroimaging Use–cases

This chapter introduces the descriptions of two neuroscience use–cases: the automatic brain segmentation, and a robust measure of changes applied to brain structures by the Alzheimer's disease. These use–cases make extensive use of image processing algorithms. Their heterogeneous source, combined to complex nature made them suitable candidates for a representation, and enactment following the scientific workflows paradigm explained in chapter 4.

5.1 MRI neuroimaging at a glance

Neuroimaging techniques have changed the way neuroscientists address questions about structural and functional anatomy, specially in relation to behavior, clinical disorders, or diseases like cerebro–vascular, neoplastic, degenerative, inflammatory, infectious, etc. Functional neuroimaging is used to indirectly measure the brain functions (e.g., neural activity), whereas structural neuroimaging deals with the brain compartments identification (e.g., shows contrast between different tissues). Among other imaging modalities such as computer tomography (CT), positron emission tomography (PET), and single photon emission computed tomography (SPECT), the magnetic resonance imaging (MRI) became largely used due to its low invasiveness, lack of radiation exposure, and relatively wide availability.

Anatomy of the brain

The central nervous system (CNS) includes the brain, protected by the skull, and the spinal cord, protected by the vertebrae. The CNS is immersed in the cerebro–spinal fluid (CSF) which is a solution acting as a buffer for the cortex, providing also a basic mechanical and immunological protection to the brain inside the skull.

The human brain consists of three main structures (see Figure 5.1):

1. The **cerebrum**. It is the largest part of the brain, and it is divided into two hemispheres (left and right). Its surface, named the central cortex, is composed of six thin layers of neurons (gray matter) which sit on top of a large collection of white matter pathways. The cerebrum directs perception, thought, judgment, decision, and imagination.

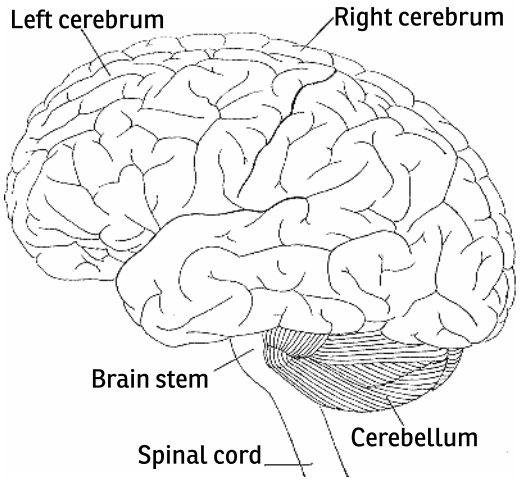


Figure 5.1: Human central nervous system. Source: *Scientific American* 199, 58

2. The **cerebellum**. It is found at the base of the brain and its composition is similar to the cerebrum. The cerebellum is the part of the CNS that regulates sensory perception, coordination and motor control.

3. The **brain stem**. It is the lower part of the brain, creating the link between the cerebral cortex, white matter and the spinal cord. It contributes to the control of breathing, sleeping and blood circulation.

The gray matter (GM) and the white matter (WM) are components of the brain, as it is shown in Figure 5.2. The GM consists of nerve cell bodies or neurons, and glial cells. It has a gray color because of the capillary blood vessels and the neuronal cell bodies. The WM is composed of nerve fiber (axons) covered up by myelinated nerve cells. The GM treats the nervous information in order to create response to the stimulus whereas the WM cells connect gray matter areas of the brain to each other, carrying on nerve impulses between neurons.

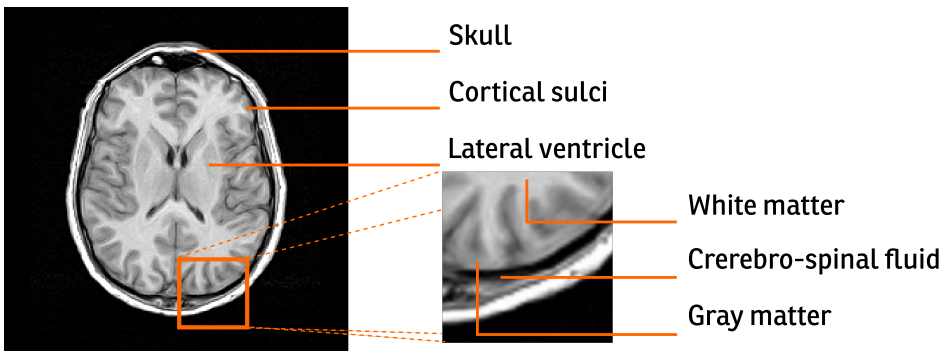
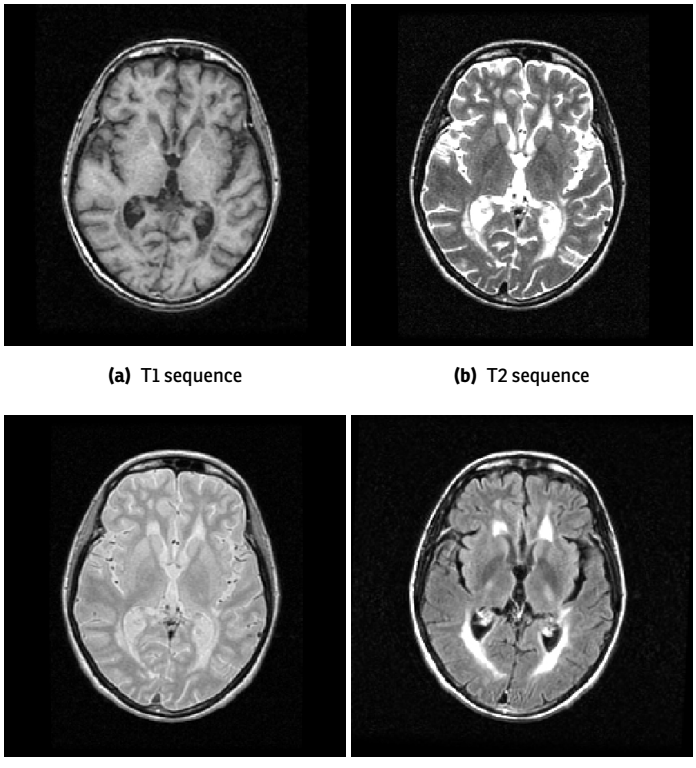


Figure 5.2: Brain tissues visualization on MRI

5.1.1 Magnetic resonance imaging

Magnetic resonance imaging (MRI) is a medical imaging technique based on nuclear magnetic resonance (NMR) used in radiology to visualize detailed internal structures. The physical phenomenon was described by Bloch et al. [1946] and Purcell et al. [1946]. The technique was then refined by Lauterbur [1973]. MRI makes use of the property of NMR to image hydrogen protons which are found in water molecules inside the human body. Thus, these protons may be assimilated to small magnets. In practice, a patient is placed in an electromagnetic field in order to displace the spin of protons from their steady state. Then, after passing of an electromagnetic wave with the resonance frequency, protons tend to return to their steady position. This *relaxation* generates another electromagnetic wave which is measured. This measure corresponds to the time of relaxation of the signal. The time depends on the intensity of the field and the nature of the tissue [Liang and Lauterbur, 1999].



(a) T1 sequence **(b)** T2 sequence

(c) DP sequence **(d)** T2-FLAIR sequence

Figure 5.3: MRI sequences from differences modalities

An MRI is processed as an image in three dimensions. That is to say a matrix in 3D on which values are assimilated to the intensity. In these 3D images, a voxel is the smallest volume unit, analogous to a pixel in 2D images. MRI provides good contrast resolution between the different soft tissues of the body, which makes it especially useful to image the brain. A typical MRI examination consists of a set of sequences, each of which are chosen to provide a different type of information about the subject tissues (Figure 5.3). For example, with particular values of the echo

time (TE) and the repetition time (TR), which are basic parameters of image acquisition, a sequence on a T1–weighted scan, water– and fluid–containing tissues are dark and fat–containing tissues are bright. The reverse is true for T2–weighted images. Damaged tissues tend to develop edema, which makes a T2–weighted sequence sensitive for pathology, and generally able to distinguish pathological tissue from normal tissue. With the inclusion of an additional radio frequency pulse and additional manipulation of the magnetic gradients, a T2–weighted sequence can be converted to a fluid attenuated inversion recovery sequence (FLAIR), in which free water is now dark, but edematous tissues remain bright. The FLAIR sequence is used to suppress CSF so as to bring out hyperintense lesions. By carefully choosing the inversion time T1, the signal from any particular tissue can be suppressed as well. In the same way, a proton density–weighted (PD) image can be produced by controlling the selection of scan parameters to minimize the effects of T1 and T2.

The remaining of this chapter describes two neuroimaging pipelines. Resulting from the NeuroLOG project, they represent a contribution of the presented work.

5.2 Automatic brain segmentation

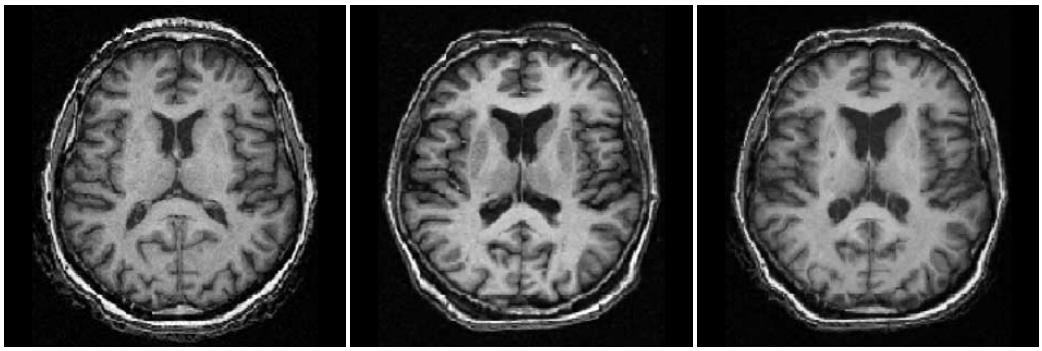
In this section is described the process of automatic segmentation of brain tissues, developed at the Asclepios Research Project,[1] towards the detection of multiple sclerosis lesions [Dugas-Phocion, 2006]. Automatic brain segmentation is suitable in neuroscience for diagnosis purpose. In particular, this method consists in a pretreatment of images for system robustness followed by the brain segmentation. It begins with a normalization of images (spatially and in intensity) and the skull–stripping. Afterwards, the segmentation into different healthy compartments classes is performed using a statistical algorithm. The method works under the assumption of a consistent database of patient's image. The input dataset is composed of multi–spectral MRI sequences T1, T2, PD and images from a reference atlas of the brain. The resulting outputs includes the binary classes and partial volumes.

5.2.1 Spatial normalization

The simultaneous use of different multi–modal MRI sequences implies to align them in the same reference frame (i.e., *registered*). T2 and PD sequences are acquired simultaneously and therefore intrinsically co–registered (i.e., they are in the same reference frame). This is not the case of T1 which also has higher resolution. The difference of reference frame is explained by the fact that sequences are not acquired at the same time. To correct the variations a registration method is used computing the displacement between two images and registering them in the same reference as shown in Figure 5.4. Different kind of registration methods exist. They either use geometric pattern to find correspondence between the images, or the intensity of the voxels [Hill et al., 2001].

In this pipeline, a rigid registration of T1 on T2 sequence is performed using the Baladin algorithm [Ourselin et al., 2000]. The algorithm considers T2 as a reference image (fixed) and T1 as a floating (moving) image. The output will be the transformation T, which transforms T1 frame into

[1] Asclepios Research Project: http://www-sop.inria.fr/asclepios/



(a) Reference image **(b)** Target image **(c)** Image after registration

Figure 5.4: Example of a RMI image registration

T2 frame, and the image T1', which is aligned with T2. The whole process follows from an iterative scheme where, at each step, two successive tasks are performed. The first stage consists in finding for each block of the floating image, the most similar sub–region in the other image, using a similarity criterion which depends on the nature of the images. The second stage consists in finding the global rigid transformation which best explains most of these local correspondences. This is done with a robust procedure which allows up to 50% of false matches. Besides its simplicity, this method provides a robust and efficient way to rigidly register images in various situations. This shows a significant improvement of the robustness, for a comparable final accuracy. Although it is more expensive in terms of computational requirements compared to other methods.

5.2.2 Atlas registration

The probability of each voxel to belong to one of the healthy tissue compartments is needed in further steps of the pipeline. The process of segmentation is based on a statistical analysis of voxels in the multi–sequence space. The atlas of the Montreal Neurological Institute[2] (MNI) [Evans et al., 1992] provides such probabilities. This stereotactic brain atlas provides T1, T2, PD modalities, and tissue probabilities maps, illustrating the standard patient.

In order to use the atlas, subject images have to be in the same reference frame. The registration is performed using the Baladin algorithm. However, since the subject images do not fit perfectly the images of the atlas generating complications in the registration, a rigid registration followed by an affine registration of the atlas T2 sequence on the T2 of the subject is performed. Once the transformation matrix has been generated, it is applied to all atlas images.

5.2.3 Skull–stripping

This step extracts the intracranial space from the image. It is preferable to isolate the brain healthy compartments, as shown in Figure 5.6, from the rest of the brain images (tissues, skull, eyes, etc) because keeping all the brain may disorder the classification step. Several methods of skull–stripping

[2] MNI atlas: http://www.bic.mni.mcgill.ca/ServicesAtlases

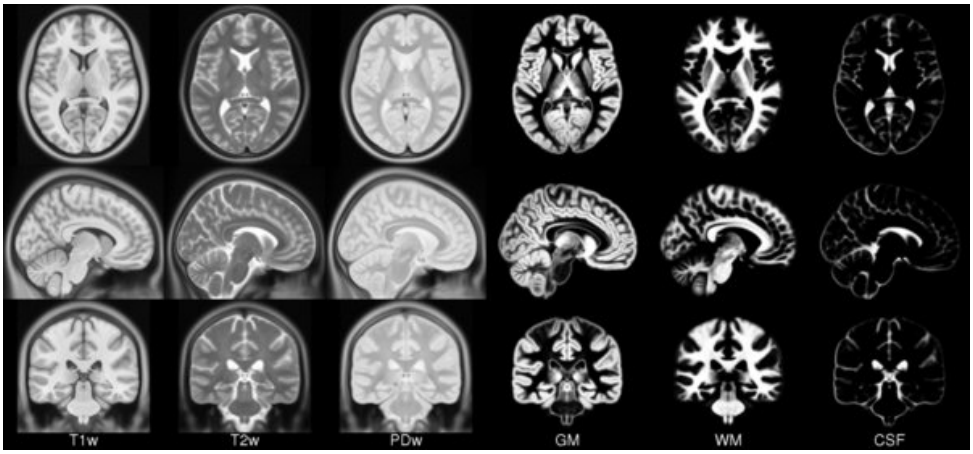
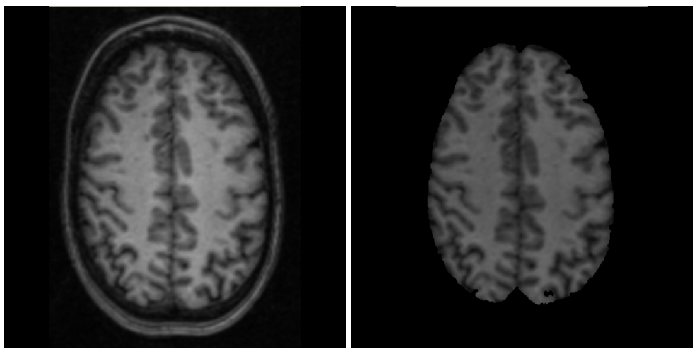


Figure 5.5: Non-linear asymmetric template of MNI atlas. Source: http://www.bic.mni.mcgill.ca/ServicesAtlases/ICBM152NLin2009

are found in the literature [Dugas-Phocion et al., 2004b]. In this case, the Expectation—Maximization method is used for the skull–stripping.

Expectation—Maximization method

The Expectation—Maximization algorithm [Dugas-Phocion et al., 2004a] is divided into two steps. First the **expectation** step corresponds to calculate the probability of each voxel belonging to each class in function of the parameters' class and a prior atlas. This step is also known as *labelization* of the image. Second, the **maximization** step consists in the estimation of the Gaussian parameters for each healthy tissue compartment class using the probabilities computed during the expectation step.



(a) T2 sequence **(b)** Skull stripped sequence

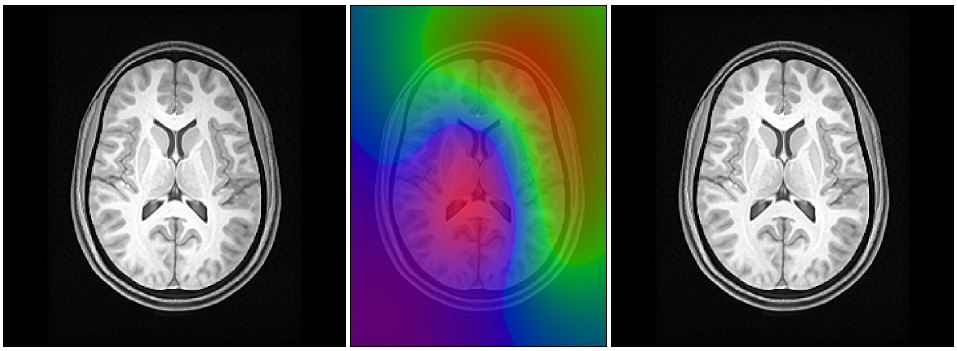
Figure 5.6: Brain skull-stripping

5.2.4 Intensity normalization

MR images are often affected by bias [Sled et al., 1998]. It means that two voxels belonging to the same brain compartment class may have different intensity. To correct this bias, a fist classification

of the brain into WM, GM, and CSF classes is preformed using the EM method with the multi–modal MRI sequences. The method consists in uniforming the intensity of the sequence for the same tissue, alternating the segmentation and bias correction. These segmentations are then used to calculate a polynomial MRI sequence, which is used to correct the bias [Prima et al., 2001] as shown in Figure 5.7.

The extraction of the bias in the T2–FLAIR sequence using the general form of the EM method does not work very well though. The low contrast WM/GM in this kind of sequences grows to use a specific spatial bias in case of T2–FLAIR. A slice–by–slice cut to alleviate the problems of the presence of bones is done. Since the segmentation includes fixed tissues, this bias is calculated in one step obtaining a simply minimization of low–frequency variations of intensity within each of the three classes WM, GM, and CSF.



(a) Uncorrected image

(b) Uncorrected image with calculated bias field superposed

(c) Bias corrected image

Figure 5.7: Brain intensity normalization. Soure: Tustison et al. [2010]

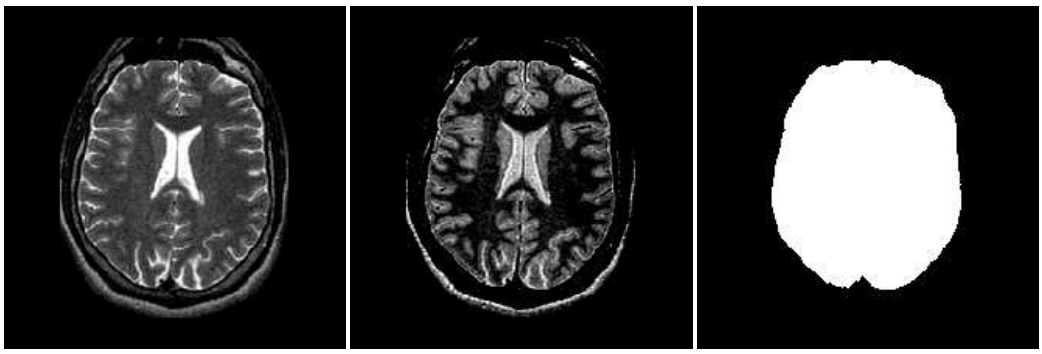
Following the treatments presented in the previous section, all images have been placed in a single spatial reference frame (i.e., the reference statistical atlas to allow the use during the labelization). The major problem at this step is to segment the images, in order to obtain a mapping of tissues.

The EM framework is used again to classify the brain MRI voxels from the unbiased sequences. In MRI the distribution of voxels intensity can be modeled by a gathering of Gaussian curves. Each brain class will be defined by a mean and a covariance matrix. Therefore, the brain tissues are divided into WM, GM, CSF, and partial volume effect (PVE) classes.

5.2.5 Brain mask segmentation

The segmentation mask of the brain is the first stages of segmentation. It is therefore essential to have a good quality of the result. The method is in fact an EM algorithm to the sequences T2/DP. In the presence of a statistical atlas, the convergence is rapid. The MNI atlas provides these probabilities. The segmentation operation just requires to move from a the probabilities of the atlas (*a priori*) to *a posteriori* probability. Operations of mathematical morphology are simple enough to

obtain a mask of brain parenchyma (neurons and glial cells). However, this operation must be conducted carefully. Two issues may be identified: over–segmentation mask in the oily areas between the parenchyma and skull, and a sub–segmentation mask. Additional operations of erosion and expansion give good results to overcome the segmentation issues. The mask is clear, even on sections of the cerebellum (see Figure 5.8. It may have some deficiencies, which will require attention in the model of tissue segmentation. This mask allows us, however, identify irregularities such as outliers, which facilitates their treatment in a statistical process like the EM.



(a) T2 sequence **(b)** DP sequence **(c)** Brain mask

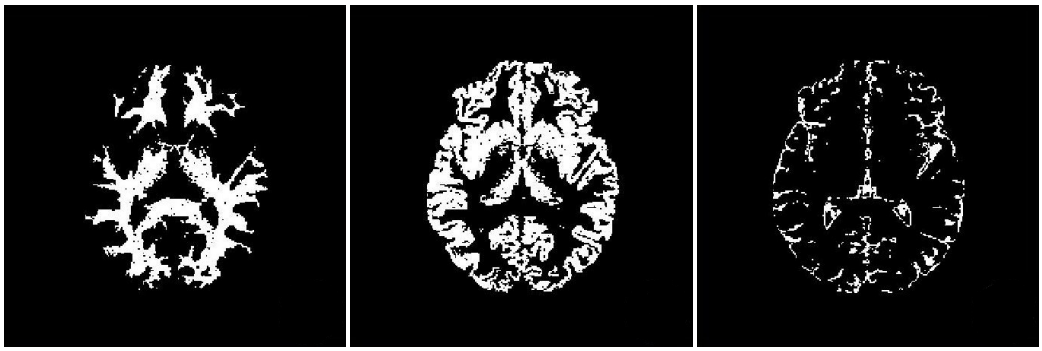
Figure 5.8: Brain mask obtained with the EM algorithm after mathematical morphology. Source: Dugas-Phocion [2006]

5.2.6 Segmentation of tissues

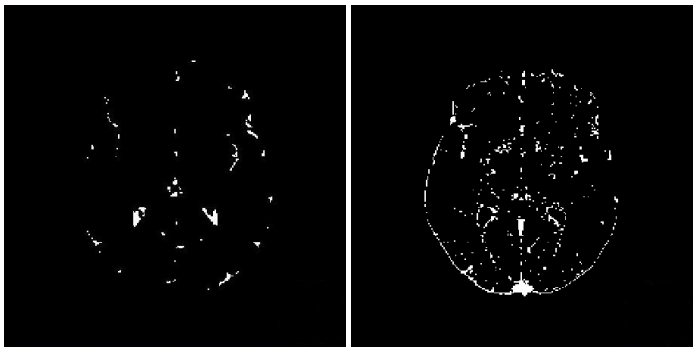
In its simplest formulation, the segmentation process takes T2 and PD sequences, in which the binary mask of the brain has been applied providing three outputs: white matter, gray matter and CSF [Dugas-Phocion et al., 2004a]. The EM algorithm gives two major results: the labeling of segmentation and the estimation of model parameters. These segmentations are illustrated in Figure 5.9.

Some assumption of uniformity of signal within the class are performed through the process of segmentation. It is established, for example, that the signal of the basal ganglia is not exactly the same as the signal of the cortex. In the same way, the image resolution is not infinite so the sample image is coupled to significant spaces in inter–tissue boundaries, especially in the cortex. This distorts the estimation classes, and invalidates the Gaussian noise model. The introduction of a partial volume model coupled to a segmentation of the vessels is used to validate the initial model. The PVE does not refer to a brain compartment. In fact, voxels of PVE are on the limit between two tissues. It means, the intensity of those voxels are a mixture of two intensities. In a brain MRI this effect appears, for example, along the limit between the CSF and gray matter.

Afterwards, MRI voxels are classified to the most probable class using the computed Gaussian parameters. This provides the segmentation of WM, GM, CSF, and PVE. During the maximization step of the EM algorithm, outliers may be detected. An outlier is a labeled voxel which Mahalanobis distance is grater than a threshold. This distance is obtained between the intensity vector (intensity of the voxel in the different sequence) of each voxel and the mean vector of each class. Finally, to



(a) White matter **(b)** Gray matter **(c)** Cerebro-spinal fluid



(d) Partial volumes **(e)** Outliers

Figure 5.9: Brain binary compartment segmentations

solve the problem of PVE and thus obtain the real segmentations of healthy brain compartments, PVE voxels are dispatched between GM and CSF in function of their intensity. All segmentations are then binarized.

In the EM method, a ratio parameter defines the faction of voxel to be used (i.e., to be labelized and then provide probabilities for the maximization step). The relation between ration value r, and the percentage of considered voxels p is given by the Equation 5.1. This parameter is important because the EM is a computationally intensive tool, so working only on a percentage of the voxel image may be interesting if, and only if, this does not affect the results.

$$p = 100 * r^{-1} \tag{5.1}$$

In the pipeline the influence of the ratio parameter, used by the EM method, is targeted to assess the final results. In fact, by taking only a part of the image voxel the speed of the algorithm is improved but it also affects the accuracy of the resulting segmentations. The study of the relationship between the percentage of considered voxels, and the trade–off between accuracy and speed is interesting for further works. To quantitatively evaluate this impact, WM segmentations are generated using different percentage of the voxel. The segmentations are compared to a reference generated with 100% of the voxels by computing their sensitivity and specificity.

Sensitivity and specificity are performance statistical measures of binary classification tests. In this case, given a segmentation of reference and a generated segmentation, *sensitivity* measures

the proportion of points segmented which belongs to the segmentation of reference and *specificity* measures the proportion of points not segmented which does not belong to the segmentation reference. Both measures are calculated using Equations 5.2 and 5.3 where a **true positive** T^+ is a voxel segmented and belonging to the segmentation of reference; a **true negative** T^- is a voxel not segmented and not belonging to the segmentation of reference; a **false positive** F^+ is a voxel segmented but not belonging to the segmentation of reference; and a **false negative** F^- is a voxel not segmented but belonging to the segmentation of reference.

$$sensibility = \frac{T^+}{T^+ + F^-} \tag{5.2}$$

$$specificity = \frac{T^-}{T^- + F^+} \tag{5.3}$$

5.2.7 Towards the detection of multiple sclerosis lesions

The processing of brain MRI in particular for monitoring patients with multiple sclerosis (MS) is useful because it is positioned as additional test in the diagnosis of this disease. It also plays a key role in monitoring the patient's condition and quantification of a response to a medication. Automatic extraction of quantifiers for multiple sclerosis has many potential applications in both clinical and pharmaceutical tests. Nevertheless, the processing of those images is difficult due to variability in size, contrast and location of lesions, so automatic segmentation of MS lesions in MRI is a difficult task. Brain compartments segmentation may be used in further step like the lesions segmentation or the evaluation of brain atrophy.

Multiple sclerosis disease

Multiple sclerosis is a nervous system disease affecting the CNS, leading to demyelination. Demyelination is the term used for a loss of myelin, a substance in the white matter that insulates nerve endings. Myelin helps the nerves receive and interpret messages from the brain at maximum speed. When nerve endings lose this substance they cannot function properly, leading to patches of scarring, or *sclerosis*, occurring where nerve endings have lost myelin. It is these areas of scarring that give multiple sclerosis its name. The characterization of MS has been done by Charcot [1872], however its causes are still unknown.

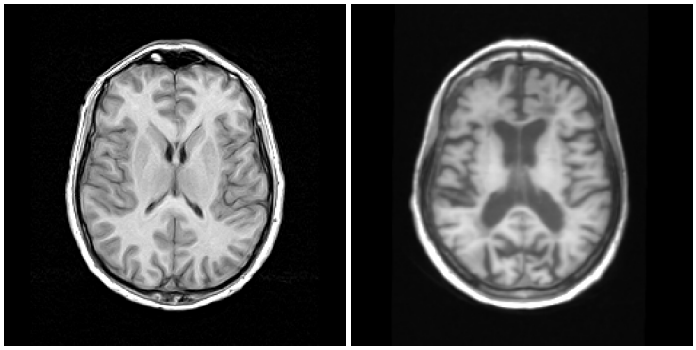
The symptoms of MS may completely vary from one subject to another because lesions may appear everywhere in the CNS. These can go from difficulty in moving to problems in speech or weakness and visual deficiencies. This is the reason of the difficulty of the diagnosis. The diagnosis of MS is done using:

- Clinical data, using visual evoked potentials to measure the speed of the brain responses.
- Laboratory data, testing the CSF to provide evidence of chronic inflammation of the CNS.
- Radiologic data, using magnetic resonance imaging to detect lesions.

Multiple sclerosis is a lifelong illness following different patterns either in discrete attacks (relapsing forms) or slowly accumulating (progressive forms). Most subjects are first diagnosed with a relapsing–remitting form which progress throws a secondary–progressive form after several years. Between attacks, symptoms may completely disappear but permanent neurological problems often persist.

5.2.8 MS lesions segmentation

In brain tissue, atrophy describes a loss of neurons, and the connections between them. Thus, the volume of WM and GM decreases in favor of CSF. In MS patients brain atrophy is identified observing larger ventricles, and cortical sulci than normal subjects (Figure 5.10). Multiple sclerosis is identified in MRI showing areas of demyelination as bright spots of the image. Indeed, visualization and position criterion of lesions in the brain have been established to determine the presence of MS [Polman et al., 2005]. MRI is superior to other imaging modalities in the imaging of demyelinating diseases because it is possible to visualize WM lesions with suitable definition (2–5 mm) and contrast resolution, and compare their progress over time. Lesions may have different shapes and localizations in the brain, hopefully they are particularly visible in the T2-FLAIR sequence where they appear with a high signal intensity. However, bony and flow artefacts are also present in the image, that is why multi–modal MRI sequences are used to isolate these artefacts.



(a) Normal T1 sequence **(b)** MS affected T1 sequence

Figure 5.10: Brain atrophy effects. Increase of ventricles and cortical sulci volumes.
Source: Database of the MICCAI'08 MS lesion segmentation challenge

The T2-FLAIR sequence is the most appropriate to visualize lesions. Different methods are available in the literature to segment these lesions. They can either be manual, semi–automatic or completely automatic [Souplet et al., 2009]. With the MRI segmentation, parameters of brain classes in the T2-FLAIR sequence are identified using he EM method. This method identifies borders of the tissues. Besides, keeping only the voxels, which have an intensity value upper than a threshold, isolates the lesions because they are hyperintense signals in that section. However, artefacts are also segmented. To isolate only lesions, a region of interest into the brain is defined. This region correspond to the WM if no lesions were present [Souplet et al., 2008].

5.3 Longitudinal atrophy detection in Alzheimer's disease

Neuro–degenerative pathologies like Alzheimer's disease (AD), is another example of neuroimaging processing. Alzheimer's disease is characterized by a co–occurrence of different phenomena, starting from the deposition of amyloid plaques and neurofibrillary tangles, to the development of functional loss and finally to cell deaths [Jack et al., 2010]. In particular, although the loss of cells is one of the final results of the pathological process taking place in the brain, it has been shown that the monitoring of structural changes provides a way to track the evolution of the disease, even at the incipient or pre–symptomatic stages [Ridha et al., 2006]. Structural MR images represent a feasible and reproducible instrument for the study of the brain's integrity. The recent availability of public studies like the "Alzheimer Disease Neuroimaging Initiative" (ADNI) [Mueller et al., 2005] provides the research of data representing the complete history of the pathological process of Alzheimer's: from the healthy condition to mild cognitive impairment (MCI), and finally to the advanced stages of the disease.

In the recent past, computational anatomy acquired increasing weight in the analysis of medical data and several methods were developed to study the brain in the *cross–sectional* (evaluating differences between different subjects) and *longitudinal* (evaluating changes in time from serial data of the same subject acting as his own control) settings. While the cross–sectional approach highlights the main differences between clinical groups, the longitudinal perspective is more useful in detecting the subtle changes related to the biological processes. A consistent integration of the longitudinal approach into a group–wise analysis represents the final goal for the development of a comprehensive model of disease evolution.

The non–rigid registration aims to measure the anatomical differences (like atrophy) between pairs of images as local geometric differences, and has been widely used in the past for the measurement of local and global anatomical changes [Boyes et al., 2006]. However, most of the present approaches are based on the assessment of image–to–image changes, a 3D problem, while the study of measurements on time–series was less explored, possibly due to the historical difficulties to collect large longitudinal dataset. Most importantly, the consistent evaluation of changes across serial images is a fundamental requirement to gain in stability and robustness of the measurements, as well as in higher accuracy in detecting biological phenomena like pathological trends.

In this section is described, step by step, a robust framework to evaluate the changes of patient's brain in time, also developed at the Asclepios Research Project [Lorenzi et al., 2010]. The pipeline develops a computationally efficient framework for the registration of serial MRI data providing a stable longitudinal atrophy measurements.

5.3.1 Time series alignment

Initially, the reorientation matches the images to the orientation of the standard template images of the FSL software library[3] [Smith et al., 2004]. It requires that the image labels are correct. It is not a registration method, so it will not align the image to standard space, it will only apply 90, 180

[3] FSL: http://www.fmrib.ox.ac.uk/fsl/

or 270 degree rotations about the different axes as necessary to get the labels in the same position as the standard template.

Given a time series $I_0, \ldots, I_n$ of raw MRIs belonging to a specific subject, the first step consists in the rigid alignment of the follow–up sequence $I_1, \ldots I_n$ to the baseline I_0, and in the re–sampling the series into a reference space for the subsequent analysis. Linear registration is an important component of structural and functional brain image analysis. It removes the spatial variability due to the differences in translations and rotations among the different scans.

The framework uses the Flirt algorithm [Jenkinsonn et al., 2002], which robustly registers the images by maximizing their correlation ratio. For each image I_i in the time series, the final affine registration matrix is obtained by composing the longitudinal rigid transformation M_i, which matches I_i to I_0, to the subject–to–template transformation M_0^T, computed by affinely registering the baseline T_0 to the reference space provided by the anatomical MNI atlas.

5.3.2 Bias correction

Magnetic resonance signal intensity measured from homogeneous tissue is seldom uniform. Rather it varies smoothly across an image. This intensity nonuniformity is usually attributed to poor radio frequency coil uniformity. gradient–driven eddy currents, and patient anatomy both inside and outside the field of view. The performance of automatic segmentation techniques which assume homogeneity of intensity can be significantly degraded due the impact of the intensity variations. Therefore, and approach a means of correcting this issue is essential for such processing.

The used approach to correcting the intensity nonuniformity [Tustison et al., 2010] does not requires a model of the tissue classes. Described as nonparametric nonuniform intensity normalization (N3), the method is independent of pulse sequence and insensitive to pathological data that might otherwise violate model assumptions. To eliminate the dependence of the field estimate on anatomy, an iterative approach is employed to estimate both the multiplicative bias field and the distribution of the true tissue intensities. This pre–processing step is central for the stability of the subsequent analysis, such as the brain mask segmentation and the non–rigid registration.

5.3.3 Baseline brain mask estimation

Since all the longitudinal changes are evaluated with respect to the baseline image, an accurate probabilistic segmentation of the baseline brain mask is required. After the initial brain extraction from the image [Smith, 2002], the probabilistic tissue segmentation (gray matter, white matter and CSF) is performed in order to obtain a probabilistic mask of the brain.

The method first removes non–brain tissue using a combination of anisotropic diffusion filtering, edge detection, and mathematical morphology [Shattuck et al., 2001]. The image is compensated for non–uniformities due to magnetic field inhomogeneities. The local estimates are computed by fitting a partial volume tissue measurement model to histograms of neighborhoods around each estimate point. The measurement model uses mean tissue intensity and noise variance values computed from the global image and a multiplicative bias parameter that is estimated

for each region during the histogram fit. Voxels in the intensity–normalized image are then classified into six tissue types using a maximum a posteriori classifier. This classifier combines the partial volume tissue measurement model with a Gibbs prior that models the spatial properties of the brain. Finally gray and white matters are combined to obtain the mask of the image.

5.3.4 Non–linear registration: the Demons algorithm

The anatomical changes between the baseline and the follow–up images are evaluated through non–rigid registration. The non–rigid registration aims to describe the anatomical differences between the pairs of images I_0 and I_i by looking for the deformation φ which maximizes their similarity. The deformation field represents a local measure of changes at the voxel level, and can be integrated in region of interest to provide a measure of the regional (global) volume change.

The non–rigid registration is derived from the log–Demons algorithm [Vercauteren et al., 2008]. In the standard log–Demons algorithm, the deformation field is given by the minimization of the sum of squared difference (SSD) between the intensities of the two images. However, the SSD is usually very sensitive to the intensity biases and does not represent a robust measure of changes. In order to avoid spurious intensity variations for morphological differences, the local correlation coefficient criteria (LCC) proposed in [Cachier, 2002] was integrated in the Demons algorithm. Given a fixed image I and a moving image J, the deformation field φ required to match the two images is computed by minimizing voxel–wise a functional which accounts for local additive and multiplicative scaling factors for the intensities. In this way the registration automatically estimates local spurious intensity differences and provides a more robust assessment of the anatomical changes.

5.3.5 Measure of the brain changes in time

This step aims to consistently measure the longitudinal changes in the time series of images by implementing a 4D registration algorithm based on the temporal regularization of the estimated deformations [Lorenzi et al., 2011b].

It relies on a hierarchical construction:

- *Spatial registration.* The deformations ϕ_i , $i = 1 \ldots n$, are estimated to match each image I_i to the baseline I_0 (brain mask estimation before).
- *Temporal regression.* The spatial deformations are used to estimate a subject–specific temporal trajectory for the longitudinal changes, for example by using a linear model in time on the deformation space.
- *Spatio/Temporal registration.* The temporal trajectory is then reintroduced in a second registration procedure, and is used as prior to drive the re–estimation of the deformations at each time point. The temporal trajectory introduces the information on longitudinal progression.

Thus, the final series of deformations are estimated by taking into account both spatial and temporal variations and will then provide a more stable and regular estimation of the longitudinal anatomical changes.

5.3.6 Quantification of the longitudinal brain atrophy

The quantification of the amount of warping applied at each voxel by the dense deformation field is usually derived from the Jacobian matrix J of the deformation in terms of the determinant. This is an average measure of *volume change*. Moreover, the Demons algorithm allows to consistently compute the flux of the vector field across surfaces (i.e., the *shift of the boundaries* required to the surface to match the homologous points during the registration process). This measure is consistent within the registration framework and is mathematically equivalent to the integration of the log–Jacobian determinant in the region of interest [Lorenzi et al., 2011a]. The framework provides both the measures of longitudinal changes evaluated in the brain mask, as well as the spatial maps of the brain's local anatomical changes, that can be used for further analysis and statistical assessment of the group–wise changes.

5.4 Summary

The use–cases presented in this chapter represent examples of the current efforts of the neuroimaging community towards a better comprehension of brain illnesses and their treatments. Their heterogeneous source and complex nature made them suitable case study candidates because they may benefit from *jigsaw* the enactment as scientific workflow, and use of DCIs. *Jigsaw* may help to provide access to these applications as compliant Web services respecting the nature of their interface invocation including inputs/outputs, parameters, and types. In addition, they may benefit from scientific workflows because the complete processing involves the execution of independent services. From the design point of view, their definition as workflows enables a higher level of abstraction showing the interactions between applications. At runtime, the resulting service composition as workflow provides three different levels of parallelism (i.e., data, service, pipeline). The parallelism grants an efficient execution. Moreover, both use–cases may benefit from distributed computing infrastructures. The computing power and the facilities of DCIs provide resources to ensure scalable executions. Additionally, the applications involved in these use–cases are heterogeneous, in terms of execution time and memory consumption making them ideal candidates for the exploitation of models of efficient use of local resources presented in Chapter 2.

Despite the specificity of each use–case, the possibility of exhibiting each application as a service shows they can be reused in order to create new scientific workflows or replace equivalent services. For instance, several registration, or skull stripping algorithms may be tested by replacing only one processor of the workflow. A more ambitious scenario may be designed to create a common set of services or sub–workflows for pretreatment of images that later can be reused in other case studies. This second scenario underlines the advantages of scientific workflows as a software modularization approach for large–scale experimentation.

Chapter 6

Enactment of Scientific Workflows on Production Distributed Computing Infrastructures

Distributed computing Infrastructures are being increasingly exploited for tackling the computation needs of large–scale applications. Grid middleware helps users in exploiting seamlessly large amounts of computing resources. However, executing large–scale applications on DCIs faces several well–identified problems often causing poor applications performance, either underperforming execution time or complete application failure. This chapter describes the methods and results obtained with the reference implementation detailed in Chapter 3, that addresses these performance problems. Results on actual neuroimaging applications show (i) the application optimization that can be performed on complex application pipelines as scientific workflows, and (ii) the impact of a production environment while performing a large–scale experiment campaign.

We assume the execution of the use–cases detailed in previous chapter as scientific workflows. The workflows are composed of multiple activities with inter–dependencies which define ordering constraints at execution time. The input datasets are composed of a large number of independent images, thus implying a high level of data parallelism. The workflow activities are fired multiple times for each data segment and the execution tests are done on production environments conditions.

In particular, we are interested in optimizing the performance of workflow enactment taking advantage of DCIs. On the other hand, we also address four issues dealing with large–scale distributed applications enactment:

1. **Low reliability** of the infrastructure causing high failure rates [Dabrowski, 2009; Huedo et al., 2006]. The larger the system used and the number of computation tasks manipulated, the more likely a failure. Failures may cause sever performance loss and in some cases stop completely the execution of an application.

2. **High latency** of computing tasks submitted to production batch systems causing low performance [Lingrand et al., 2009b]. The splitting of an application computation logic in many tasks lends towards more parallelism but the gain may be easily compensated by the time needed to handle all tasks generated in a competitive production batch system. In the case

of workflow–based applications with inter–dependencies between tasks, the sequential submission of tasks to long–queue batches will be highly penalizing.

3. **Unfair balance** between shorter and longer computation tasks [Isard et al., 2009]. The very complex tuning of large–scale submission systems, involving meta–brokers and many schedulers, makes extremely difficult to achieve fair balance between short and long tasks in a computation process. The larger the computing time discrepancy between tasks, the higher the impact.

4. **Complex deployment & scalability** of distributed computing applications [Krishnan and Bhatia, 2009]. Beyond middleware parametrization, the deployment of services may have a strong impact on application performance as servers easily become overloaded in large–scale runs. Appropriate deployment is also the key to achieving good scalability.

The chapter is structured as follows. First, the principles to design the scientific workflows of both case studies are introduced. Then, materials and methods for the experimentation are described. We later present the experimental results. Finally, a discussion derived from the experiments is developed, focusing on the four issues previously mentioned.

6.1 Workflow design

The two neuroimaging use–cases described in Chapter 5 have been enacted as scientific workflows. Both are represented in Figures 6.1 and 6.2 respectively. This section details an example of service composition involved in the process of building a workflow.

The pipelines of automatic brain segmentation and longitudinal atrophy detection in Alzheimer's disease from sections 5.2 and 5.3 are described as scientific workflows for enabling their enactment using MOTEUR. Their services have been linked together and iteration strategies have been selected, according the definitions of Section 4.1.2, to produce the appropriate dataflow for processing the inputs.

Each service input has been composed with iteration operators. For example, in the case of the rigid registration of the automatic segmentation workflow shown in Figure 6.1, data concerning patients has been composed with a dot product to avoid cross–road composition, and then composed with a cross product with other data. Tags have been used in the inputs to refer images of the same patients. Considering the registration of $T1$ on $T2$ sequence, three different cases are possible with patients A and B, and a configuration file including the execution *parameters*:

1. All inputs are identified with the same tag. So they are composed by dot products. In this case, with inputs $\{T1_A, T1_B\}$; $\{T2_A, T2_B\}$ and $\{parameters_A, parameters_B\}$ the results from the composition are:

$$\{T1_A, T2_A, parameters_A\}; and$$
$$\{T1_B, T2_B, parameters_B\}.$$

2. Only the images $T1$ and $T2$ are identified with a tag. So they have to be composed by a dot product and then are composed with a cross product with the input parameters. In this case, with inputs $\{T1_A, T1_B\}$; $\{T2_A, T2_B\}$ and $\{parameters\}$ the results from the composition are:

$$\{T1_A, T2_A, parameters\}; and$$
$$\{T1_B, T2_B, parameters\}.$$

3. All inputs are composed by cross products. In this case with inputs $\{T1_A, T1_B\}$; $\{T2_A, T2_B\}$; and $\{parameters_A, parameters_B\}$ the results from the composition are:

$$\{T1_A, T2_A, parameters_A\}; \{T1_A, T2_A, parameters_B\};$$
$$\{T1_A, T2_B, parameters_A\}; \{T1_A, T2_B, parameters_B\};$$
$$\{T1_B, T2_A, parameters_A\}; \{T1_B, T2_A, parameters_B\}; and$$
$$\{T1_B, T2_B, parameters_A\}; \{T1_B, T2_B, parameters_B\}.$$

These configurations have been used to test different values of the ratio parameter of the EM service. Indeed, for the first invocation of EM in the skull stripping, we combine common parameters to all patients. It means, the second composition case was performed with dot and cross products tagging the images but not the parameters. Whereas, for the following EM invocation, in the classification performed before the bias estimation, the ratio parameters vary for each patient therefore just a dot product is performed. Acting this way allows users to put two times the same patient but with two different file parameters.

Initially, we completed the design of this first workflow using the SCUFL language of Taverna [Oinn et al., 2004] workflow manager. The workflow does not requires high level abstraction of data composition because it is executed using the same datasets requiring only parameters modifications. Therefore, simple iteration strategies are required. On the other hand, the enactment of the longitudinal atrophy detection in Alzheimer's disease requires more complex iteration strategies due to the number of services inputs. Moreover, the workflow composition in this second use–case involves the treatment of several patients at the same time. It is necessary to take into account the modification of parameter as well as complex data composition. Thus the resulting workflow requires design elements as the *flat cross* product provided only by the GWENDIA language. Beyond this composition requirement differences the design of both workflows includes the same construction steps: wrap the CLI tools as services using the *jigsaw* wrapper, deploy the applications as Web services, and finally compose the services by means of the MOTEUR workflow manager.

6.2 Materials and methods

This section details the experimental conditions in terms of the execution environment. This description is followed by the definition of the reference evaluation measures in order to evaluate the application optimization of the automatic brain segmentation case study, and the performance for the Alzheimer's disease use–case.

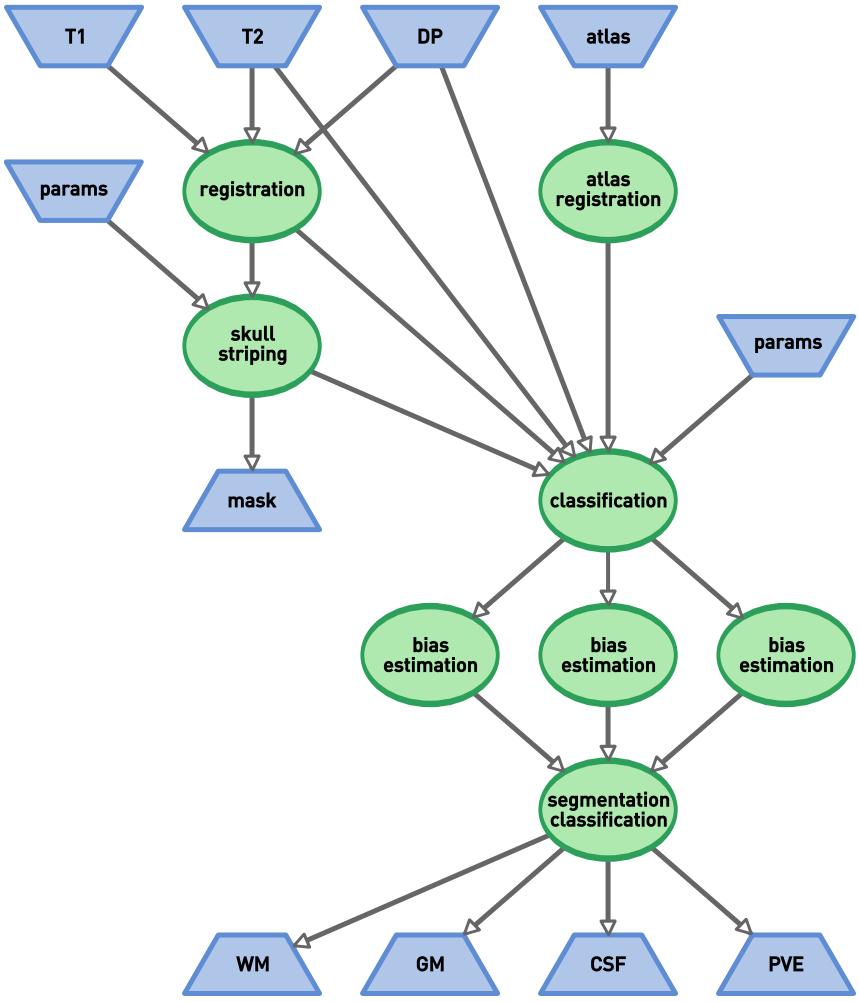


Figure 6.1: Simplified schematic representation of the automatic brain segmentation workflow where ellipses represent services and trapezoids represent input/output data

6.2.1 Execution environment

We performed the experiments using the framework detailed in Section 3.3.2. We use a server with 2 quad-core processors at 2.67 GHz and 16 GB of memory for local executions. This resource is used to implement the decision model described in Section 2.2 in combination with the European Grid Infrastructure detailed bellow. To complete the framework we also take advantage of the DIRAC pilot jobs management system which improves experiments performance.

European Grid Infrastructure

The European Grid Infrastructure (EGI) is a collaborative effort involving more than 10,000 users over 50 countries. Its objectives are to enable a sustainable production infrastructure of resource providers; to support structured international research; to manage virtual organizations; and to

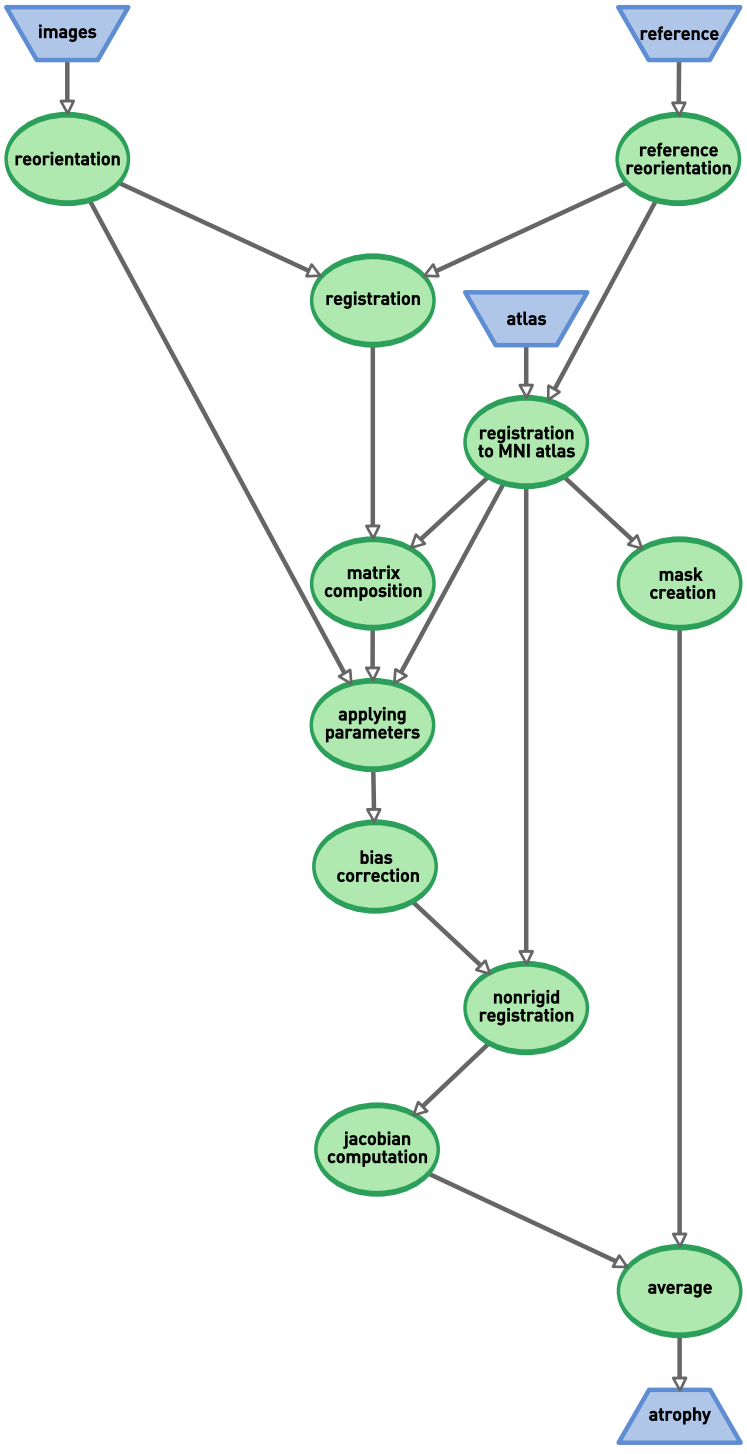


Figure 6.2: Simplified scientific workflow representation of atrophy rate from longitudinal analysis at Alzheimer's disease where ellipses represent services and trapezoids represent input/output data

provide middleware and training services through the federation of national and domain specific resource providers [Newhouse, 2011]. The infrastructure includes in excess of 300 sites offering around 340,000 processor cores, and more than 100 Petabytes of storage. The infrastructure is

available to users around the world achieving a sustained workload of half a million computer tasks or jobs every day.

DIRAC

The Distributed Infrastructure with Remote Agent Control (DIRAC) project is a complete Grid solution for a community of users needing access to distributed computing resources.[1] It is designed to be a generic data management and job submission system providing means for managing tasks on Grid resources taking over the workload management functions. The DIRAC architecture consists of numerous cooperating distributed services and light agents built within the same framework following the Grid security standards.

DIRAC introduced the now widely used concept of pilot jobs. Pilot job is a type of multilevel scheduling, in which a resource is acquired by sending pilots before and then the application can schedule work into that resource directly, rather than going through a local job scheduler which would lead to queue waiting time for each work unit. Pilot jobs are most often used on systems that have queues to avoid multiple waits during scheduling. Pilot jobs allows DIRAC to build an efficient Workload Management Systems optimized in a central task queue.

6.2.2 Measures of evaluation

Application optimization

We are interested in reproducing the work performed by neuroscientists while experimenting to underline the advantages of working with the proposed experimental framework and EGI. A service–oriented approach execution based on a workflow enactment offers the possibility of automatize manual tasks such as data staging and scripting development, reduce potential errors of data and execution management, and improves the final execution timespans. These advantages have an impact during experimentation because more detailed experiments can be performed with the same datasets or early conclusions can be validated with larger–scale executions. We focus in the qualitative results without measuring aspects related to the infrastructure performance. Specifically we try to evaluate parameters of an experiment in order to optimize the application.

Scalability performance

Another set of experimental trials aims at quantifying the latency endured by, and the speedup of the entire workflow. We are mainly interested in the average latency $\bar{x}$ of all job submissions, and the final workflow execution timespan for the speedup S calculation. In addition, the execution failure rate and the maximum number of theoretical concurrent executions of submitted jobs are also studied as they are important indicators in the scalability analysis. Three execution types are considered:

1. **Execution on grids**. The workflow is executed by submitting jobs directly to the WMS. This is the default behavior when working on the Grid.

[1] DIRAC: http://diracgrid.org/

2. **Multilevel scheduling execution**. The workflow is executed using DIRAC. This represents a basic environment considering pilot jobs.

3. **Efficient execution**: The workflow is executed implementing the decision model for efficient use of local resources defined in section 2.2 in combination with the *multilevel scheduling execution*.

6.3 Results

Experiments have been designed to validate the approach modeled in Chapter 2 and the resulting reference implementation detailed in Chapter 3. The framework proposed is stress tested using real applications related to the treatment of brain conditions. In case of the automatic brain segmentation, an application optimization is performed presenting results of a typical execution of a neuroimaging study. We are interested to tune the parameters of the EM execution in order to obtain a valid threshold range of the ratio parameter for a suitable brain segmentation. In case of the atrophy rate measurement from the longitudinal analysis at Alzheimer's disease, we focus on the qualitative evaluation of the execution platform to estimate the performance improvement while using a production DCI as EGI.

6.3.1 Results on application optimization

The workflow of the automatic brain segmentation uses twice the EM algorithm to perform the skull stripping and the classification of tissues for the bias estimation. The ratio parameter of the EM service is evaluated according its sensibility and specificity according to Equations 5.2, and 5.3. The EM step consists in the estimation of the Gaussian parameters for each healthy tissue compartment class. These assessments are computed from the voxels intensities of the MRI. A ratio parameter define the fraction of voxel to be used (e.g., if the ratio is equal to 1 then all voxels are considered). In this part, we use the percentage of considered voxels.

The experiment assesses the influence of the ratio parameter on the workflow results. In fact, by taking only a part of the image voxel, the speed of the algorithm could be improved but it could also affect the accuracy of the resulting segmentations. Therefore the relationship between this parameter and the compromise between accuracy and speed is studied for use in further works. To quantitatively evaluate this impact, WM segmentations have been generated for different ratios and have been compared to a reference segmentation (i.e., segmentation with ratio equal to 1) by computing the algorithm's sensitivity and the specificity.

It is important to underline that voxels are chosen randomly in the 3D image. Consequently, different results can be obtained for a same ratio parameter. To minimize the influence of this randomization, many executions have been done and mean values of the sensitivity and the specificity have been computed. Figure 6.3 displays these values as a function of the percentage of voxel considered with the variations around mean values. For this application, the power of the grid provides an efficient help to generate all the results (9 executions per ratio value). Indeed, the ratio parame-

ter was written in an input parameter text file and has been assimilated as a relative to the patient. Acting this way allows us to test all the different ratio parameters with each patient's MRI.

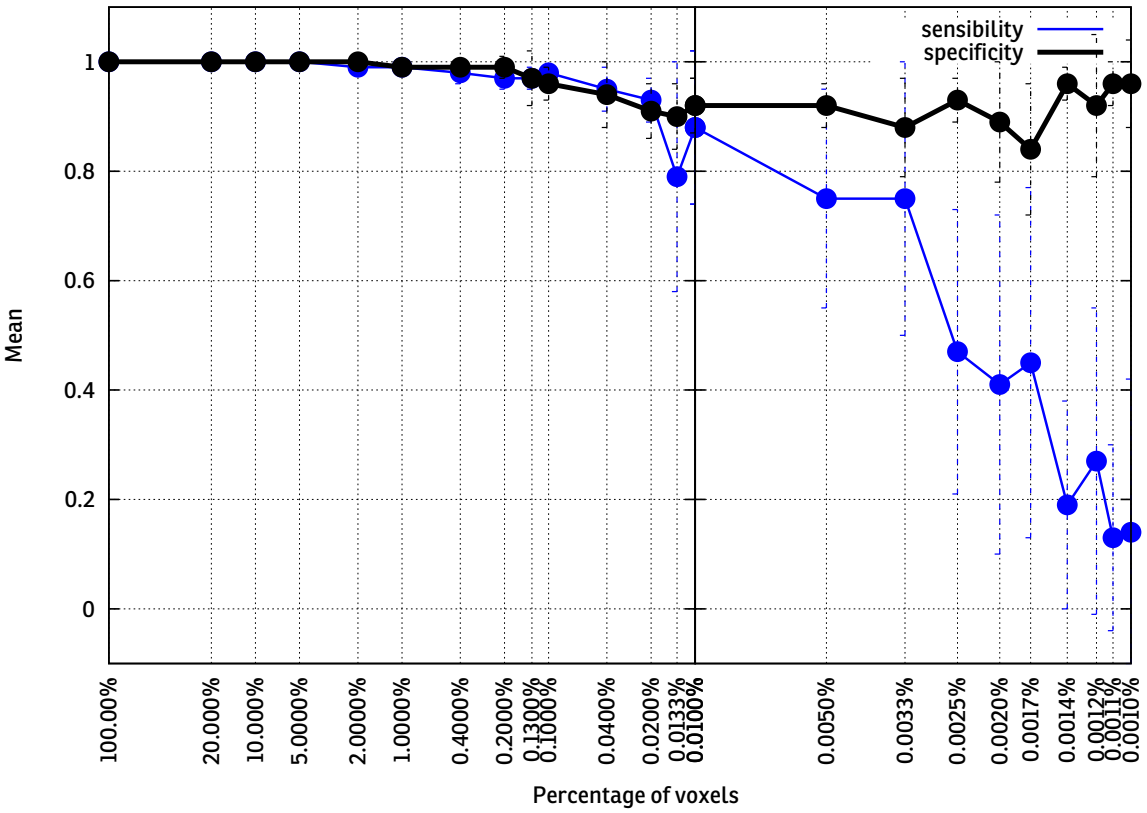


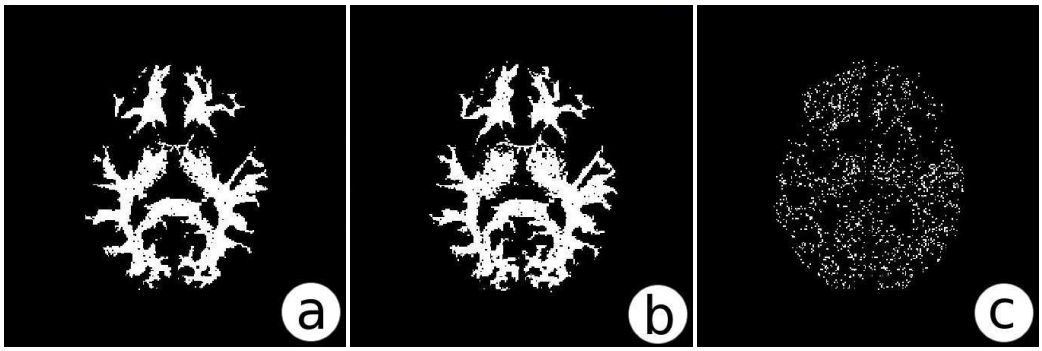
Figure 6.3: Mean sensibility and specificity of white matter segmentations in function of the percentage voxel

Due to the skull–stripping step, the segmentation of the different healthy compartments is done on approximately 830,000 voxels. On Figure 6.3, we observe that the sensibility is decreasing while the percentage of voxels considered is decreasing. The specificity is more stable but those two quantities are increasingly variable. Taking less than 1% of the voxels in our algorithm leads to results with too high variability: we cannot accept that different execution (with random voxel selections) lead to different results.

First, in this case, a WM segmentation with a specificity of 100% would mean that each voxel defined as belonging to (resp. not to) the white matter is really belonging to (resp. not to) the white matter in the segmentation of reference. But this doesn't mean that our segmentation results are accurate for low percentage ratio. Indeed, in our case, specificity and accuracy should not be confused because there are far more true negatives (voxels out of brain) than true positives (voxels really belonging to WM).

Secondly, the drastic decrease of the sensibility means an increase of the number of false negative which corresponds to the voxel really belonging to the WM but not labeled as such. This reveals that after a certain threshold value of the ratio, there are not enough voxels any more in order to be able to define the Gaussian class parameter from the class estimation step of the EM. Finally, these

results reveal that using only 1% of the voxels of the image in the EM method would divide its execution time by 3 or 4 (compared to the execution with 100% of the voxels), without impacting the WM segmentation quality (Figure 6.4).



(a) 100% **(b)** 2% **(c)** 0.02%

Figure 6.4: White matter binary segmentation from the workflow for different ratio percentage values

6.3.2 Results on scalability performance

The workflow execution on a production environment as EGI is confronted to a constant workload variation of the infrastructure. Therefore, is not possible to reproduce similar conditions between different executions. Moreover, these executions are performed indistinctly on computer elements with different performance capacities resulting in some cases very different execution timespans. These are the reasons to be interested in the variations of the submission latency and the overall workflow speedup instead of absolute values of the latency and final timespans.

The longitudinal atrophy detection in Alzheimer's disease workflow is a good example to test scalability, to validate the decision model presented in Section 2.2, and to evaluate the execution environment setup. Services composing the workflow shown in Figure 6.2 are heterogeneous in terms of average execution time and memory consumption as shown in Table 6.1. A benchmark of the average execution time of each service on the target DCI was previously done to estimate the values of t_i, r_i, and T_{Max} required in Equations 2.1 and 2.2.

Several patients could be processed in parallel without performance loss assuming availability of resources on the DCI. For each experiment type, the workflow was executed with patients datasets which size grows exponentially from 1 to 256 (see Table 6.2), and with 2 to 5 images associated to each patient. This leads to an average of 25 service executions per patient. The experiments were performed using inputs of the Alzheimer's Disease Neuroimaging Initiative (ADNI) database.[2]

Latency

The average latency in minutes ($\bar{x}$), the standard deviation (σ), the minimum (min) and maximum (Max) registered latencies, the median absolute deviation of the latency (MAD), the range ($Max -$

[2] ADNI: http://adni.loni.ucla.edu/

Services	Average time [min]	Memory [MB]
images reorientation	1.450	150
reference reorientation	1.450	150
rigid registration	3.217	250
registration to MNI atlas	4.183	250
matrix composition	1.333	150
applying parameters	2.317	200
bias correction	7.167	500
mask creation	14.350	1,000
nonrigid registration	174.783	6,500
Jacobian computation	3.300	1,000
average	1.333	150

Table 6.1: Benchmark of average services execution on EGI

Patients	Concurrent Services	Total Executions
1	5	35
2	10	70
4	16	108
8	32	216
16	64	432
32	123	824
64	243	1,624
128	481	2,852
256	962	6,416

Table 6.2: Summary of services executions

min), and the interquartile range (IQR) are calculated for the three workflow executions modes as it is shown in Table 6.3. We are mainly interested to the values of the MAD and IQR because they are robust statistics that are not affected by outliers. In the context of executions on EGI, such outliers have exhibited a high impact on latency due to load variability [Lingrand et al., 2009a] making difficult the interpretation of the execution results.

We observe in graphically in Figure 6.5 a sustained increase of the latency when the input dataset size increases in grid executions. The values of σ and Max increase as well in all types of executions as shown in Table 6.3. This behavior is expected as the increasing number of jobs loads the WMS submission queues and DIRAC when using pilot jobs. For instance, in Table 6.2 we show the increment from 25 concurrent executions for one patient up to 962 executions for 256 patients. Conversely, the multilevel and efficient optimization methods reduce the average latency significantly due to the reuse of worker nodes passing by the scheduler mechanisms obtained with pilot jobs. Focusing on the largest dataset runs, we verify that the latency is lower for a same number of patients with multilevel scheduling than with grid execution, and even lower with efficient execution showing the relevance of execution without waiting times thanks to the use of local resources. This

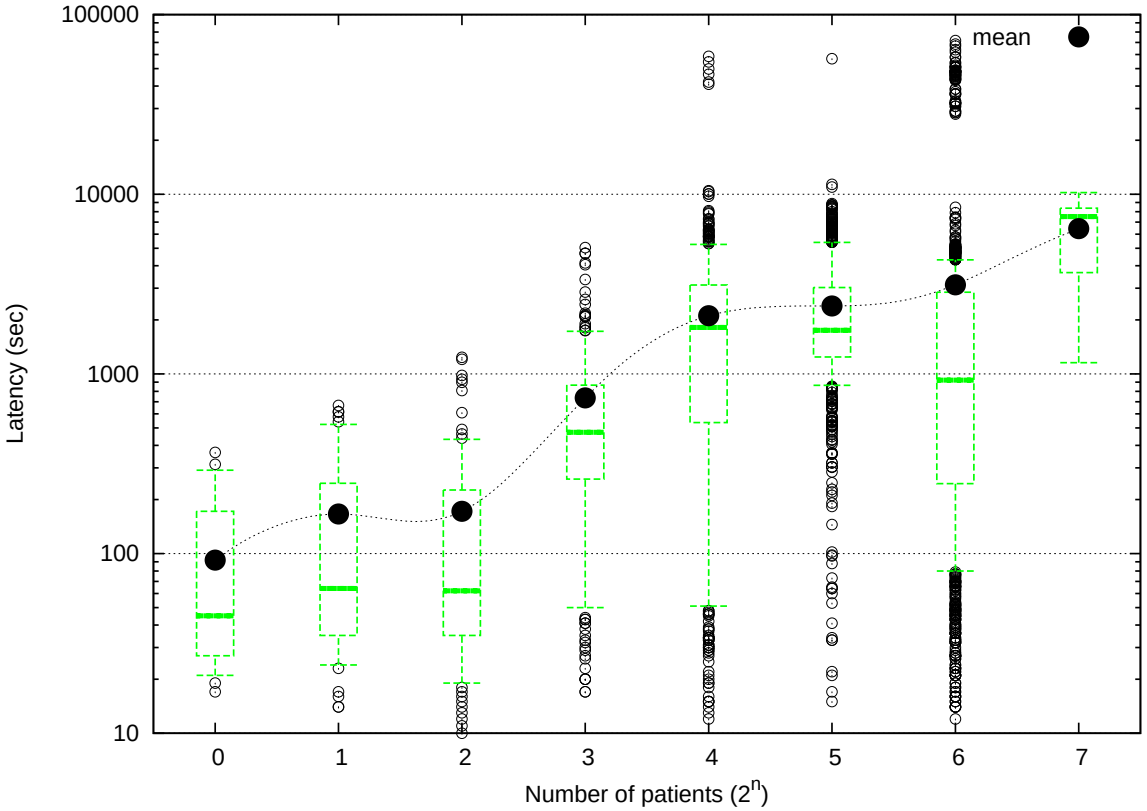


Figure 6.5: Latency variability on grid executions

behavior is verified with the MAD, a variability measure comparable to the standard deviation. The MAD shows the dispersion reduction among all latencies when the optimization methods are implemented (see Figure 6.6). Similarly, the IQR shows a tendency for gradual increment of the range as increasing the number of patients, however the optimization methods significantly lowers these values exhibiting the attenuation effect obtained initially with the pilot jobs and then reinforced with the implementation of decision model for submission on local resources. The use of limited local resources shows up with large datasets obtaining similar IQR values in multilevel and efficient executions.

In Figure 6.8 (page 99), we present an example of the timeline diagrams for the three experiment types. Graphically, we can observe on the top of the diagram the time evolution of a saturated WMS during a Grid execution that results in all tasks having a similar waiting time delaying the invocation of last services by all accumulated latency. This latency is reduced once the multilevel method is implemented. Finally, we can observe that latency is reduced in the same proportion with the efficient execution type; even more in some cases there is no latency at all. Nevertheless, the use of local resources potentially reduces the final execution timespan as we can observe in Table 6.4.

	Patients	$\bar{x}$	σ	min	Max	MAD	Range	IQR
grid	1	1.815	1.712	0.283	6.100	0.433	5.817	2.883
	2	2.783	3.180	0.233	11.117	0.667	10.884	3.700
	4	2.871	4.049	0.167	20.617	0.675	20.450	3.284
	8	12.251	13.836	0.283	84.150	5.208	83.867	10.134
	16	35.141	33.672	0.333	164.983	11.467	164.650	28.666
	32	39.841	29.903	0.250	189.200	10.500	188.950	29.850
	64	52.237	145.353	0.200	1,194.117	13.583	1,193.917	43.484
	128	107.185	53.747	0.250	774.233	27.417	773.983	78.367
	256	178.289	101.185	0.217	1,661.483	51.008	1,661.266	100.467
multilevel	1	1.525	1.112	0.467	5.300	0.400	4.833	1.066
	2	2.049	2.354	0.425	13.409	0.504	12.984	1.367
	4	3.558	6.031	0.350	26.217	0.350	25.867	2.084
	8	2.428	2.750	0.383	12.400	0.408	12.017	2.267
	16	5.349	10.609	0.289	93.011	0.880	92.722	3.867
	32	10.017	24.844	0.284	174.142	1.138	173.858	5.142
	64	6.637	8.637	0.242	84.842	1.846	84.600	9.825
	128	14.134	17.741	0.175	161.517	7.896	161.342	19.350
	256	26.293	40.736	0.100	349.783	8.269	349.683	32.389
efficient	1	0.304	1.019	0.000	4.372	0.000	4.372	0.000
	2	0.582	1.954	0.000	9.350	0.000	9.350	0.000
	4	0.477	1.349	0.000	7.117	0.000	7.117	0.000
	8	0.460	1.152	0.000	7.067	0.000	7.067	0.000
	16	1.559	5.174	0.000	52.975	0.000	52.975	0.950
	32	5.470	17.212	0.000	121.125	0.171	121.125	3.463
	64	6.205	11.644	0.000	52.767	0.900	52.767	6.000
	128	10.279	16.120	0.000	193.667	1.517	193.667	15.350
	256	24.730	48.920	0.000	393.467	2.900	393.467	29.217

Table 6.3: Latency statistics in minutes for all execution modes

Speedup

Three different types of speedup are considered to evaluate the impact of the execution framework on application performance as shown in Table 6.4. The *traditional speedup* S is defined as the ratio of a reference, the sequential running time of the application, over the timespan measured during a parallel run. The speedup measures the improvement with regard to the total CPU consumption that may vary significantly between computing elements. In addition, we determine the *workflow speedup* $S_w = p \times T_1/T_p$ where p is the number of patients, and T_i is the workflow execution timespan for i patients in a given execution mode. S_w measures the global workflow improvement rather than execution time. The S_w shows the speedup evolution within an execution mode as a function of the number of patients (and datasets). Finally, the *relative workflow speedup* S'_w is computed as S_w but taking T_1 of the *grid execution* type for all cases. The value of S'_w shows the execution improvement with regard to a constant reference of executions on the Grid and represents a good comparator between execution modes.

We observe for all execution types in Table 6.4 that the speedups is effective from one patient ($S > 1$). The increasing speedup demonstrates all levels of parallelism (i.e., data, service, pipeline)

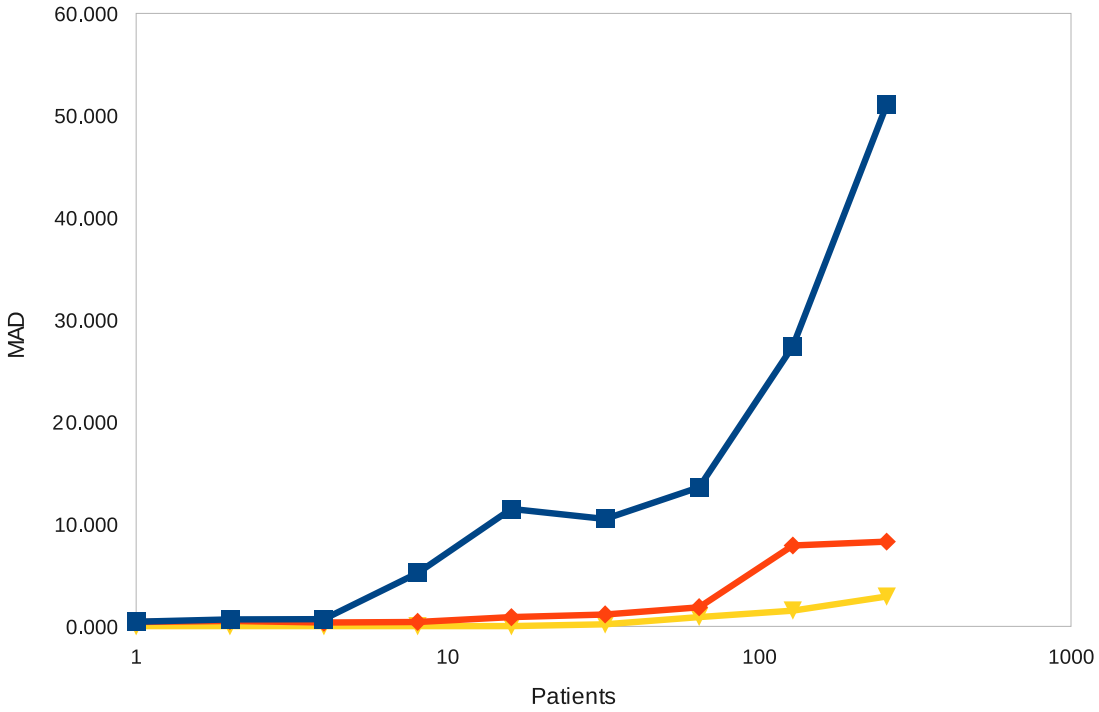


Figure 6.6: MAD of average latency. Grid execution in blue, multilevel execution in red, and efficient execution in yellow

implemented with the workflow enactment. The speedup increases significantly even if the latency increases showing the success of the resources acquisition on the DCI. The workflow enactment enables concurrent executions improving the final execution timespan, specially in case of pilot jobs use for large numbers of patients.

Figure 6.7 shows the *workflow speedup* evolution for each execution mode. We can observe that pilot jobs play an important role in the speedup improvement. Moreover, the higher values of S_w in the efficient execution verifies the cumulative effect of including pilot jobs, and the use of the submission decision model. Although, this behavior is marginal with large number of patients because of the limited number of local resources.

According to the results, while the implementation of the optimization methods improves the speedup, the final execution timespan may not differ significantly between the implementation of the *multilevel execution* and *efficient execution*. Similarly, the number of failures has the same order of magnitude across the executions using pilot jobs due to the heterogeneity of the computing elements on EGI. It means that even if the use of local resources attenuates the failure rate it does not represent a safeguard to reduce the final execution timespan but its use has a clear influence on latency in absolute terms (MAD and IQR). In fact, the almost constant failure rate present along all reported experiments is due to several factors on the production environment, namely full storage elements, temporal unavailability of middleware services such as the file catalog server or the proxy certificates manager, unexpected timeouts while storing data, or specific applications errors resulting of incompatibilities with OS computing elements and/or missing system libraries.

	Patients	Timespan [hours]	Total CPU [hours]	Failure rate	S	S_w	S'_w
grid	1	8.024	14.037	0.00%	1.749	1.000	1.000
	2	6.556	20.047	25.00%	3.058	2.448	2.448
	4	7.326	29.651	14.29%	4.047	4.381	4.381
	8	14.394	83.656	31.86%	5.812	4.460	4.460
	16	21.144	223.438	17.46%	10.567	6.072	6.072
	32	22.442	358.608	27.77%	15.979	11.441	11.441
	64	33.619	572.193	11.31%	17.020	15.275	15.275
	128	35.863	1,328.756	14.83%	37.051	28.639	28.639
	256	41.531	2,388.036	11.36%	57.500	49.460	49.460
multilevel	1	3.382	11.631	0.00%	3.439	1.000	2.373
	2	4.569	21.448	10.26%	4.694	1.480	3.512
	4	4.484	37.047	1.82%	8.262	3.017	7.158
	8	4.478	69.354	2.26%	15.488	6.042	14.335
	16	6.200	104.168	1.51%	16.800	8.727	20.706
	32	8.614	227.472	2.02%	26.407	12.564	29.808
	64	12.831	698.307	13.71%	54.423	16.869	40.023
	128	20.528	1,160.384	9.09%	56.527	21.088	50.033
	256	19.959	1,857.050	1.99%	93.043	43.379	102.918
efficient	1	3.152	10.155	2.78%	3.222	1.000	2.546
	2	3.574	20.263	1.41%	5.670	1.764	4.490
	4	3.461	32.010	0.00%	9.249	3.643	9.274
	8	3.354	56.137	0.46%	16.737	7.518	19.139
	16	4.048	115.145	0.92%	28.445	12.458	31.715
	32	7.750	219.229	1.02%	28.288	13.015	33.131
	64	9.560	388.227	6.13%	40.610	21.101	53.717
	128	12.536	962.033	6.95%	76.742	32.184	81.930
	256	18.655	2,255.662	7.42%	120.915	43.254	110.112

Table 6.4: Timespan statistics for all execution modes

In summary, these quantitative results comply with the behavior of a production DCI as EGI that is reported in literature reporting a dynamic working load, and heterogeneous resources availability. The implementation of the decision model and the optimization methods show a significant reduction of invocation latency, and an execution speedup improvement. It is important to notice the independence between the use of local resources and the data size used as input. It means, the local execution may also involve large amounts of data even if these are processed by a short execution job. At the same time, the storage elements are considered distributed across the infrastructure therefore, the execution using the efficient model it is not associated in any case to local storage.

6.4 Discussion

A lot of research efforts have been invested in dealing more or less independently with the four well–known issues of large–scale infrastructures mentioned in the introduction chapter (low reliability, high latency, unfair balance, complex deployment and scalability). We face them with

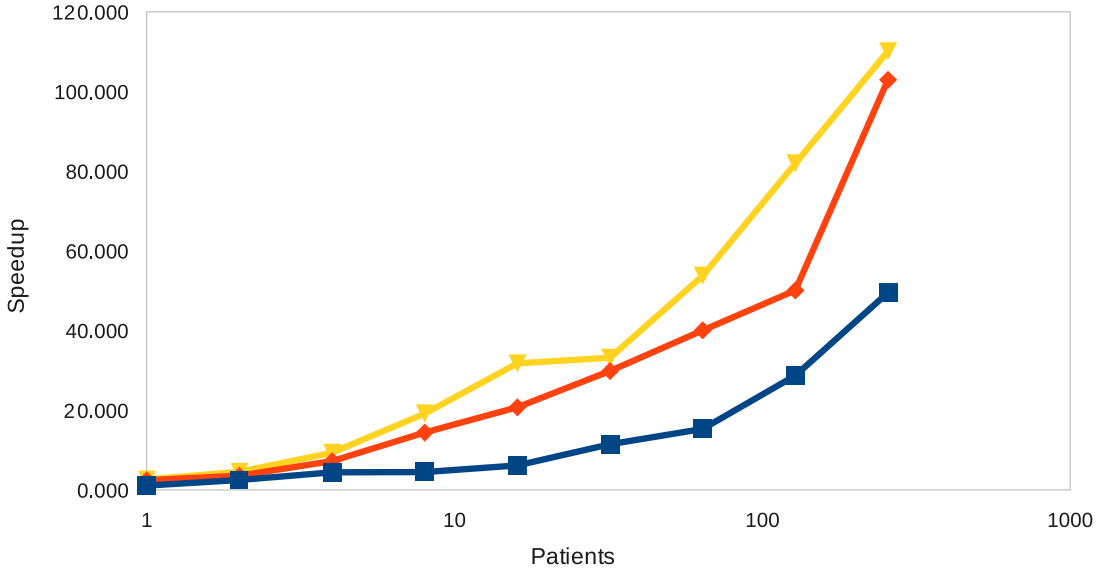


Figure 6.7: Execution speedup as a function of the number of patients. Grid execution in blue, multilevel execution in red, and efficient execution in yellow

the design of an end–to–end execution framework in which SOA principles are adopted to enable the execution of distributed workflow applications on large–scale datasets. Using an SOA approach allows users to scale the execution of their applications, and flexibly extend the execution framework according to their computation needs. Besides, in SOA various optimization strategies can easily be integrated to improve the application performance as shown through experiments campaign reported here. Following we resume how we tackled (and verified during experiments) each problem as part of the goals of the thesis.

Failure recovery. Networking and computing infrastructures are subject to random resource failures. The likeliness of failures increases with the number of physical entities, as seen in large–scale distributed systems today [Dabrowski, 2009; Huedo et al., 2006]. Recovering from failures becomes a critical issue to improve the reliability of the infrastructure, preventing the correct completion of many application runs. Numerous works addressing this issue have been proposed in the literature including the check–pointing, live migration [Kangarloun et al., 2009; Koslovski et al., 2010], job replications [Casanova, 2007] and submission strategies [Lingrand et al., 2009b]. On general purpose production infrastructures, job resubmission is often the only general failure recovering solution available, as check–pointing and migration usually either make restrictive assumptions on the computational processes or they require application instrumentation. The final makespan could be increased, specifically with longer applications, but resubmission ensures that the application execution can always continue and finish successfully. This approach is implemented in the framework by controlling the status of submitted jobs and defining a resubmission policy when a failure occurs.

Lowering latency. The splitting of an application's computation logic in many tasks lends towards more parallelism but the gain may be easily compensated by the time needed to handle all tasks generated in a competitive production batch system. In the case of a workflow–based application with inter–dependencies between tasks, the sequential submission of tasks to long batch queues will be highly penalizing. Addressing the high latency issue, many works study multiple submissions approaches [Subramani et al., 2002; Casanova, 2007; Lingrand et al., 2009b]. The results of these studies confirm that submitting tasks several times increases application performance. However, users who do not use multiple submission are penalized. Furthermore, without considering the capacity of batch schedulers, high number of submissions can overload the batch schedulers and then degrade the overall system performance.

Alternatively, pilot jobs systems help users in reserving a pool of computing resources during the execution of the application [Casajus et al., 2010], being considered as a bridge between batch systems and systems supporting resources reservation. A pilot job is submitted to a workload manager to reserve a computing resource. User jobs are then pulled from the job queue to computing nodes by successfully started pilot jobs. Each pilot job can thus process sequentially several user jobs without introducing delay between two of them. Each pilot is subject once to the workload manager queuing time but the jobs they process are not. Another advantage of pilot jobs to the classical submission approach include the sanity checks of the running environment before assigning resources for execution. They also allow users to create a virtual private network of computing resources reserved for executing their tasks, and they implement effectively the pull scheduling paradigm. Our execution framework extensively uses pilot jobs reducing latency and making executions more reliable because broken resources are filtered by the pilot jobs.

Task fairness. The very complex tuning of large–scale submission systems, involving meta–brokers and many schedulers, makes it extremely difficult to achieve fair balance between short and long tasks in a computation process. Yet, production infrastructures are not only used for long running jobs processing data–intensive applications but they are also frequently used for processing shorter jobs. Statistical results shows that more than 50% of the jobs take less than 30 minutes for execution [Isard et al., 2009]. While the high latency has less impact on long running jobs, short jobs are heavily penalized if they have long waiting times before execution. The larger the computing time discrepancy between tasks, the higher the impact. Users therefore require a mechanism of resource fair sharing to avoid that long jobs monopolize the whole computing resources, and delay the completion of other users (short) jobs.

Pilot jobs also improve handling of short jobs as they reduce individual jobs queuing time. However, although dedicated to a specific user, pilot job systems usually do not implement fairness among the user's jobs and pilots may be overloaded by the processing of longer jobs similarly to a Grid meta–scheduler. Therefore, our approach combines more dedicated resources out of a distributed infrastructure with the capacity of DCIs to improve handling of short jobs. Local resources are more reliable since the user is administrator of computing nodes, thus failures coming from the software dependencies are lowered. Executing applications locally reduces the number of job submissions remotely removing the submission phase and delays of middleware initialization. This

then reduces the waiting time of other jobs in the queue for obtaining computing resources on remote infrastructures. Nevertheless, as the number of computing resources in the local server is limited, the more jobs submitted locally, the longer the execution time needed to finish all jobs. We define a decision model in Section 2.2 to decide whether a task is executed on local resources or submitted to a DCI.

Deployment & scalability. Beyond middleware parametrization, the deployment of application services may have a strong impact on application performance as servers easily become overloaded in large–scale runs [Krishnan and Bhatia, 2009]. Some initiatives like GASW [Ferreira da Silva et al., 2011] or LONI Pipeline [Dinov et al., 2009] propose tools to reuse scientific applications on DCIs but they have scalability limitations or interoperability constraints respectively. Concerning Web service–related projects, tools such as GEMLCA [Delaitre et al., 2005], and GRAVI [Chard et al., 2009] manage services lifecycle at different levels, enabling dynamic deployment and/or supporting of non–functional concerns. However, their adoption involves the use of an homogeneous middleware. Our execution framework relies on a legacy application code wrapper that both provides a standard Web service interface to all application computing components, and helps managing the complete lifecycle of the resulting services.

In practice multiple services containers, acting as a proxy between users and the production DCI, may be configured in the framework. Each container naturally has a limited capacity to process concurrent services. When the size of input dataset increases, the number of services submitted concurrently may exceed its capacity. The replication of servers into the system (scaling out) resolves this limitation. It increases the performance without modifying the framework architecture.

Addressing all concerns together. The execution framework used during this experimental campaign addresses simultaneously the production DCIs shortcomings by combining advanced job submission strategies, services replication, and including the use of local resources during workflow enactment. The implementation of job resubmission improves the reliability by instantiating a system capable of error overcoming from remote executions. Then the adoption of pilot jobs for multi-level scheduling ensures the reduction of latency. Pilot jobs represent a new approach to overcome long queues of batch schedulers reusing computing resources efficiently. In order to tackle the unfair balancing resulting from the competition of short/lightweight application tasks with the long/heavyweight ones, a decision model dispatching tasks among local and remote resources is implemented. The deployment of services provides transparent mechanisms of applications reallocation, over local and remote resources, holding back technical details far from final users. Finally, the scalability heedfulness ensures large–scale experiment campaigns by enabling services resiliency.

The delivery of an integrated execution environment is eased by the application of SOA principles, made possible by the workflow formalism used to model distributed applications. SOA has been adopted to a large extent in middleware design [Foster, 2006]. For instance, the Swift workflow management system [Zhao et al., 2007] provides an integrated working environment for job

scheduling, data transfer, and job submission. It is built on top of a uniform implementation based on Globus toolkit. Yet, production infrastructures hardly ever comply to a homogeneous middleware stack, nor adopt a single communication standard for all core and community services. Conversely, traditional workflow management systems like Taverna [Oinn et al., 2006a], or Triana [Taylor et al., 2005] support service invocation enabling interoperability but they do not natively execute code on DCIs. In our architecture, both middleware and application components are deployed as services. The application code is instrumented non invasively to comply to this model through a Web service builder aware of DCIs computing capability. Using an SOA approach allows users to scale the execution of their applications and flexibly extend the execution framework according to the computation needs.

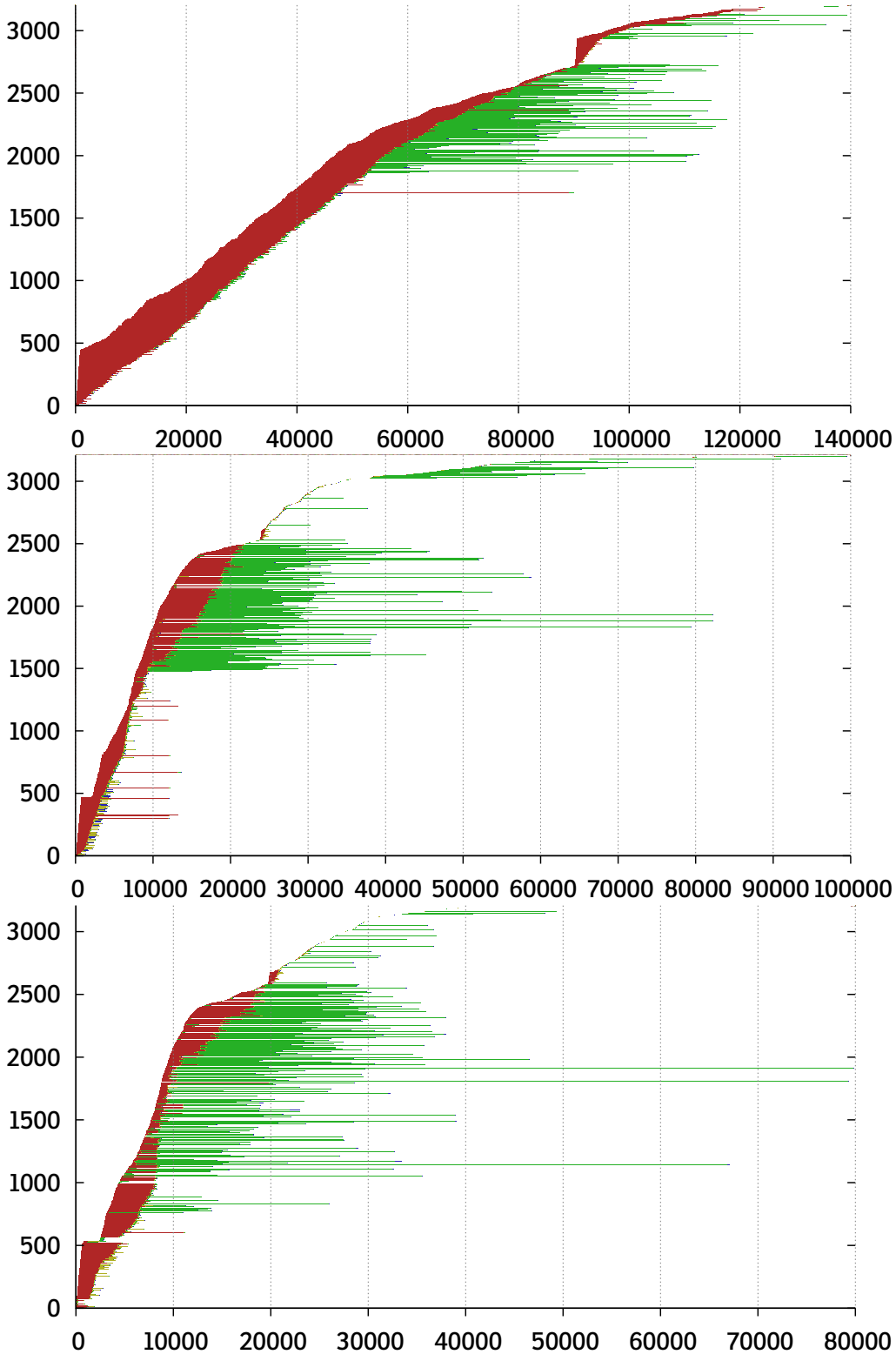


Figure 6.8: Timeline diagrams (execution time [seconds] as a function of the number of services) for the workflow executions of 128 patients on EGI. From top to down: grid execution, multilevel execution, and efficient execution. Each service is represented by a horizontal bar composed of two parts: the first (in red) is the latency time between submission and acquisition of a computing resource, the second (in green) is the execution time including data transfers.

Conclusions and Perspectives

Conclusions

This work aims at improving the Grid users experience, specially among the neuroimaging scientific community, by simplifying experiments and enacting scientific applications efficiently.

The need for a uniform and generic characterization of legacy applications led to a formal abstraction of CLI tools to describe their invocation, taking into account the execution context. It provides a set of definitions to identify details concerning invocation arguments. This abstraction enables the definition of the application interface independently from the configuration required for execution. In addition, several infrastructure configuration environments can be included within the definition, supporting the execution in heterogeneous platforms. The abstraction of CLI applications, defined in chapter 2, is used in a reference implementation framework to expose those tools as services.

The study of the dataflow and computing requirements of two neuroimaging use–cases, described in chapter 5, led to model them as scientific workflows composed of services in a data–driven approach. A coordinated work with neuroimaging experts was needed to understand completely the details of applications involved in complex pipelines and transform them into scientific workflows. We took advantage of two scientific workflow definition languages to describe the data iteration strategies and control structures of the dataflow. In the case of the automatic brain segmentation, we defined the workflow using the SCUFL language. This first composition revealed the limitations of SCUFL when working with high data dimensionality. Therefore, the second use–case, a measure of changes applied to brain structures by the Alzheimer's disease, was designed with the GWENDIA language. This later definition enabled the representation of a more complex dataflow. In both cases, the use of MOTEUR led a proper enactment on heterogeneous platforms.

We identified the requirements to enact the designed workflows and we highlighted the need to fulfill non–functional concerns associated to work with sensitive data in a distributed working environment. *Jigsaw*, an extensible framework integrating external concerns, was developed in the context of two driver projects to enable the execution of services that compose the workflows. This reference implementation manages the complete lifecycle of services from creation stage to invocation, passing through deployment and instrumentation on the executing infrastructure. Some salient characteristics of the framework include the implementation of a CLI application wrapper as Web service and a generic consumer API; an embeddable GUI in third–party software; and the extension support for new execution strategies on alternative distributed infrastructures. This frame-

work not only represents a proof–of–concept of the feasibility of our approach. Nowadays, *jigsaw* is successfully used in production environments by external research projects.

We adopted a service–oriented approach during the implementation of the *jigsaw* framework to ensure scalable and flexible experimentation. The implementation takes advantage of service containers enabling execution scalability. The use of an open and production tested container, such as Apache Tomcat, eases the services management with mechanisms like hot deployment. It also grants more advances procedures of server–side administration such as clustering, load balancing and servers farming. The possibility of a transparent deployment of several containers ensures to serve a larger number of service invocations. In addition, the implementation of a compatible service–oriented framework based on Web standards extends the number of potential users by providing interoperable interfaces. In fact, the access to heterogeneous platforms through a single standard–based interface simplifies the experimentation of scientists.

In addition, we handled some strategies to ensure reliable and resilient executions on heterogeneous infrastructures. The use of DIRAC, a workload managed system supporting multilevel jobs scheduling execution by the instrumentation of "pilot jobs" on the target infrastructure, allows us to tackle high failure rates commonly found in production environments such as EGI. The performance loss of job submission was significantly reduced in combination with two more pragmatic solutions: job resubmission policies and data replication. The modular implementation of *jigsaw* offered the possibility to integrate DIRAC into the framework improving the executions extensively. On the other hand, the implementation of resubmission policies was resolved as a simple extension without compromising the rest of the framework. We also noticed the beneficial effect of data replication in order to avoid storage servers unavailability that can completely stop a workflow execution.

Finally we defined a workflow–oriented model to exploit efficiently local and remote resources. The adoption of production DCIs ensures access to a large number of computing resources. However, it often implies high latencies and failure rates. In the case of scientific workflows, the latency and failures are amplified because of the enactment of several levels of parallelism. Moreover, the unfair balance of short and long–term executions makes extremely difficult to achieve an optimized execution due to the very complex tuning of large–scale submission systems. Conversely, the use of local resources offers reactive and more reliable resources but often brings resource limitations with large experimentation. The definition and implementation of a simple and effective decision model to combine both types of resources showed a non negligible global improvement in terms of failure rate and submission latency reduction.

The conceptual contributions and development presented throughout this work addressed simultaneously CLI applications lifecycle management in distributed environments, and production DCIs shortcomings. The implementation of a reference standard–based framework, combined to the adoption of efficient mechanism to overcome resource unavailability, and the formal definition of a decision model for job submissions provide an alternative to tackle issues of data intensive experimentation.

Perspectives

Towards an open and standard description of CLI applications

Many specifications to describe CLI applications execution have been presented in the literature. A majority of them share the same principles and concepts. Some abstractions are also geared towards specific fields of science like bioinformatics [Senger et al., 2008] or chemistry [Krishnan et al., 2009]. However, these initiatives are not consensual efforts, thus the specifications are likely ignored or inadvertently recreated outside of their original purpose. Moreover, they do not include systematically a formal data schema making it difficult to fully exploit the features of the abstraction.

The abstraction of CLI applications developed in this work may be the starting point for an enhanced version. The experience gained from interactions with neuroscientists, and the feedback received during the development of the *jigsaw* framework resulted in some potential modifications and extensions towards an open and standard specification for the description of CLI applications interfaces and execution contexts. New resolution models (e.g., cross references between arguments, or support for arguments that are input and output simultaneously) and mappers in the declaration of arguments, may introduce a cleaner and more flexible description of tools. The definition of additional schemes for domain–specific tools support may also promote a uniform description of CLI applications for better dissemination. This kind of extensions may benefit to semantic abstraction of applications as well, because the resulting descriptions can be directly reused for the generation of semantic annotations or the definition of ontologies. The integration of execution environment configuration in the description would represent an additional contribution to the standardized abstraction of CLI applications description addressing heterogeneous infrastructures.

Forthcoming development paths

As part of the permanent goal of improving the user experience, multiple development paths are considered. First, we can integrate JSAGA [Reynaud, 2010], an open initiative of the Grid community, to enable a uniform data and execution management across heterogeneous infrastructures. This integration would reduce the management of multiple sources of libraries and dependencies. The use of JSAGA can also lead the reuse of *jigsaw* components as JSAGA adapters (i.e., programmatic interfaces designed to minimize coding effort for integrating support of new technologies) to release features of our framework like the result processing as data–typed collections. In the same direction as the adoption of solutions that support a wide range of technologies, we can integrate natively DIRAC through its APIs allowing users to aggregate, in a single management system, resources of different nature. This represents the inclusion of a new layer into the framework supporting the infrastructure management. The inclusion may simplify the implementation of the framework removing the need of extending new executors for additional infrastructures. Furthermore, DIRAC may bring to the framework additional control and auditing mechanisms for distributed infrastructures.

Second, planning the executions of applications as scientific workflows, we can enhance the interfaces of the framework to ensure a modular cohesion with workflow management systems. In fact, one goal of the European SHIWA project is to address the fine–grained interoperability execution of workflows [Plankensteiner, 2010]. Fine–grained interoperability focuses on the transformation of workflow representations in order to achieve workflows migration from one system to another. The power of the fine–grained workflow interoperability stands in exploiting the most appropriate enactor for a certain workflow application, independently from the language in which it was created. The use of technological–neutral mechanisms to provide access to application executions would definitely reduce the integration effort between application execution and workflow enactment. The *jigsaw* framework becomes an interesting tool for achieving that goal since the first–class entities in scientific workflows are services, and they must be interoperable to enable the execution on different DCIs. Part of this initiative is already achieved with the integration of the *jigsaw* API in the MOTEUR workflow engine.

Complementarily, we are convinced that we can make an important improvement to the *jigsaw* framework joining new technological trends like autonomic computing. The framework, for example, would take advantage of ongoing efforts of self–management of distributed computing resources [Krikava et al., 2011] incorporating mechanisms to adapt the responsiveness of application executions according to the infrastructure status.

Prospects in neuroinformatics

Neuroinformatics merges the power of computational analysis with neuroscience evolving from a simple use of computers for data organization to the current development and application of sophisticated computational tools for large–scale data and image management, analysis and modeling of brain function. This discipline continuously searches for methods that facilitate new insights through the integration and analysis of large and diverse datasets.

Scientific workflows become a potential catalyst to transform the way experimental campaigns are conducted in neuroinformatics. Their adoption in bioinformatics, for instance, has become a driver in the creation of a dynamic community to find, share, and exchange data, models, and processes [De Roure et al., 2009]. Yet, the use of distributed workflows, enacting services deployed over remote sites, remains infrequent in neuroinformatics. Several factors influence negatively a broader development of neuroinformatics. The cumbersome access to data due to legal policies restricts sharing. Frequently tools are not fully developed requiring long iterative process of test before reaching mature stages. Unlike bioinformaticians, only specific collaborations between research teams have been established. Moreover, there is still some reluctance to change working practices even if they are error–prone or take longer to perform large–scale experimentation.

The evolution of neuroinformatics today has to be based on a broad dissemination of existing tools and continuous development. This multidisciplinary approach involves advanced concepts and technologies that are not easy to assimilate and handle. Nevertheless, the advantages attached to the adoption of high–level abstraction applications and the automation of previous manual data-processing and analysis tasks may represent a trigger. The promotion of scientific workflows can accelerate the development of neuroinformatics by a separation of concerns between dis-

cipline–specific content and domain–independent software. Neuroscientists understand the impact of interactions between tools in the creation of analysis methods supporting disciplinary research. Therefore, the development and utilization of the *jigsaw* framework become a step forward in this context but much more effort in dissemination of concepts such as scientific workflows and development of similar frameworks is needed before initiatives like *jigsaw* may be applied routinely across many disciplines.

Outlook on service–oriented science

We strongly advocate for service–oriented science [Foster, 2005a]. This approach has the potential to increase scientific productivity by making tools available, and thus enabling the widespread automation of data analysis and computation. Service–oriented science enables publishing and accessing data and scientific applications. The definition of standard interfaces and protocols allows users to encapsulate data and applications as interoperable services. Therefore, tools formerly accessible only to restricted communities now can be made available to all. Service–oriented architectures resolve past data interchange and execution autonomy issues, and their implementations are bridged successfully to external infrastructures opening the door to scalable experimentation.

Service–oriented science takes advantage of distributed computing infrastructures enabling large–scale experimentation in remote and cross–institutional contexts. Analogously, cloud computing can also foster the development of service–oriented science. Cloud computing is a computing model providing software, middleware and computer resources on demand where the physical location, scale, and maintenance remains transparent to users. Cloud computing can be a key benefit in service–oriented science, despite the challenges that cloud computing carry on in scientific environments: external providers raising security issues, commercial strategies of business–oriented operations, or throughput computing incompatibilities. It harnesses the rapidly increasing computing power as well as virtualization technologies to create a resource delivery model "as a service" at different levels, namely, infrastructure as a service (IaaS), platform as a service (PaaS), or software as a service (SaaS). This emerging approach, adopted into the strategy of most industries nowadays, defines the concept of "elasticity" as the feature of automated, dynamic, flexible and frequent resizing of resources that are provided to an application by the execution platform. This elasticity provides dynamicity and adaptivity to the efficient experimentation and honors the service–oriented science approach.

Appendices

Schema of the CLI application description

The definition of the schema uses the RELAX-NG compact syntax [ISO/IEC 19757-2:2008]. The notation convention is as follows:

- Reserved keywords are in *italics*.
- Definitions are in sans serif.
- Values are in `monospace`.
- Data types include their namespace prefix in SMALL CAPITALS.

```
start = bundle
bundle = element bundle { interface , implementations }
interface = element interface { version , symbolicName , description? ,
    organization? , copyright? , reference? , contactAddress? , arguments? }
implementations = element implementations { implementation+ }
version = element version { XSD:NMTOKEN{ pattern = '\c+' } }
symbolicName = element symbolicName { XSD:NCName{ pattern = '\s' } }
description = element description { XSD:token }
organization = element organization { XSD:token }
copyright = element copyright { XSD:token }
reference = element reference { XSD:token }
contactAddress = element contactAddress { XSD:NMTOKEN }
arguments = element arguments { argument+ }
argument = element argument { identifier , stream , type , mapper ,
    implicitness , space , label , option? , hint? , content , nesting }
identifier = attribute identifier { XSD:ID }
stream = attribute stream { streamType }
type = attribute type { typeType }
```

```
19 mapper =  attribute mapper { mapperType }
20 implicitness =  attribute implicitness { xsd:boolean }
21 space =  attribute space { xsd:boolean }
22 label =  element label { xsd:NMTOKEN }
23 option =  element option { xsd:NCName }
24 hint =  element hint { xsd:NCName }
25 content =  element content { model , crossRef , matter? , extensions? , template? }
26 model =  attribute model { modelType }
27 crossRef =  attribute crossRef { xsd:boolean }
28 matter =  element matter { text }
29 extensions =  element extensions { extension+ }
30 extension =  element extension { xsd:NMTOKEN }
31 template =  element template { basePath & baseName & baseExtension }
32 basePath =  attribute basePath { xsd:IDREF }?
33 baseName =  attribute baseName { xsd:IDREF }?
34 baseExtension =  attribute baseExtension { xsd:IDREF }?
35 nesting =  element nesting { dimension , separator , beginCollection , endCollection }
36 dimension =  element dimension { xsd:nonNegativeInteger }
37 separator =  element separator { text }
38 beginCollection =  element beginCollection { text }
39 endCollection =  element endCollection { text }
40 implementation =  element implementation { release ,
41       platforms , configuration? , attachment? }
42 release =  element release { xsd:NCName }
43 platforms =  element platforms { platform+ }
44 configuration =  element configuration { variable+ }
45 attachment =  element attachment { xsd:anyURI }
46 platform =  element platform { infrastructure , profiles ,
47       sharedEnvironment? , sharedArtifact? }
48 infrastructure =  attribute infrastructure { infrastructureType }
49 profiles =  element profiles { profile+ }
50 sharedEnvironment =  element sharedEnvironment { variable+ }
```

```
51 sharedArtifact = element sharedArtifact { xsd:anyURI }
52 profile = element profile { job , target , boundEnvironment? , boundArtifact? }
53 job = attribute job { jobType }
54 target = element target { xsd:NMTOKEN }
55 boundEnvironment = element boundEnvironment { variable+ }
56 boundArtifact = element boundArtifact { xsd:anyURI }
57 variable = element variable { category , name , value }
58 category = attribute category { categoryType }
59 name = element name { xsd:NCName }
60 value = element value { text }
61 streamType = ( "input"
62   | "none"
63   | "output" )
64 typeType = ( "boolean"
65   | "double"
66   | "integer"
67   | "string"
68   | "URI" )
69 mapperType = ( "archive"
70   | "console"
71   | "filesystem"
72   | "regexp" )
73 modelType = "regular"
74 modelType |= ( "directory"
75   | "expansion"
76   | "replacement" )
77 infrastructureType = "single"
78 infrastructureType |= ( "egi"
79   | "g5k"
80   | "other"
81   | "pbs" )
82 jobType = "normal"
83 jobType |= ( "mpi-lam"
84   | "mpi-mpich"
85   | "mpi-mpich2" )
86 categoryType = ( "infrastructure"
87   | "internal"
88   | "system" )
```

Template–based source code generation

Apache Velocity is a Java–based template engine. It is a simple and powerful development tool that allows users to easily create and documents (source code) that format and present data. When using Velocity in an application program, the following steps are performed:

1. Initialize Velocity.
2. Create a Context object.
3. Add data objects to the Context.
4. Set a base template.
5. Merge the template and data to produce the output.

The "context object" is a common technique for moving a container of data around between parts of a system [Apache Velocity]. The idea is that the context is a carrier of data between the Java layer and the template layer. Objects, and their methods and properties, are accessible via template elements called references. The language uses references defined through *statements* to embed content in the resulting code. There are three types of references in the language: variables, properties and methods.

An statement is meant to incorporate dynamic content by replacing the reference in the template. It is identified with the "#" character. The shorthand notation of a variable consists of a leading "$" character followed by an *identifier*. The notation of a property consists of a leading "$" character followed an followed by a dot character (".") and another identifier. Finally, the notation of a method consist of a leading "$" character followed a identifier, followed by a *method body*. A method body consists of a identifier followed by an left parenthesis character ("("), followed by an optional parameter list, followed by right parenthesis character (")").

Several *directives* are defined as script elements in the template language. They can be used to creatively manipulate the output of the Java code. They include statements, conditionals, loops, and macros. The *macro* script element allows template designers to define a repeated segment of a template. They are very useful in a wide range of scenarios because they are saving keystrokes and minimizes typographic errors.

Bibliography

Apache Velocity. Apache Velocity Developer's Guide. The Apache Software Foundation, January 2010. URL https://velocity.apache.org/engine/devel/developer-guide.html.

Mark Baker, Rajkumar Buyya, and Domenico Laforenza. Grids and Grids Technologies for Wide–area Distributed Computing. *Software: Practice and Experience*, 32(15):1437–1466, 2002. doi: 10.1002/spe.488.

Bénédicte Batrancourt, Michel Dojat, Bernard Gibaud, and Gilles Kassel. A core ontology of instruments used for neurological, behavioral and cognitive assessments. In *International Conference on Formal Ontology in Information Systems*, FOIS 2010, Toronto ON, Canada, May 2010. doi: 10.3233/978-1-60750-535-8-185.

Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of Scientific Workflows. In *Workshop on Workflows in Support of Large–scale Science*, WORKS 2008, Marina del Rey (CA), USA, November 2008. doi: 10.1109/works.2008.4723958.

Julien Bigot, Hinde Lilia Bouiziane, Christian Pérez, and Thierry Priol. On Abstractions of Software Component Models for Scientific Applications. In *EuroPar Workshop on Abstractions for Distributed Systems*, DPA 2008, Las Palmas de Gran Canaria, Spain, August 2008. doi: 10.1007/978-3-642-00955-6_49.

Andrew D. Birell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984. doi: 10.1145/2080.357392.

Felix Bloch, William W. Hansen, and Martin Packard. Nuclear Induction. *Physical Review*, 69(3–4):127, 1946. doi: 10.1103/physrev.69.127.

Richard G. Boyes, Daniel Rueckert, Paul Aljabar, Jennifer Whitwell, Jonathan M. Schott, Derek L.G. Hill, and Nicholas C. Fox. Cerebral Atrophy Measurements Using Jacobian Integration: Comparison with the Boundary Shift Integral. *NeuroImage*, 32(1):159–169, 2006. doi: 10.1016/j.neuroimage.2006.02.052.

Pascal Cachier. *Recalage non rigide d'images médicales volumiques — Contribution aux approches iconiques et géométriques*. PhD thesis, École Centrale des Arts et Manufactures, Sophia Antipolis, France, January 2002.

Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the Lhcb Dirac Team. DIRAC Pilot Framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219 (1–6), 2010. doi: 10.1088/1742-6596/219/6/062049.

Henri Casanova. Benefits and Drawbacks of Redundant Batch Requests. *Journal of Grid Computing*, 2(5): 235–250, 2007. doi: 10.1007/s10723-007-9068-6.

Jean-Martin Charcot. *Leçons sur les maladies du système nerveux faites à la Salpêtrière*, volume 1. A. Delahaye, Paris, France, 1872.

Kyle Chard, Wei Tan, Joshua Boverhof, Ravi Madduri, and Ian Foster. Wrap Scientific Applications as WSRF Grid Services Using gRAVI. In *International Conference on Web Services*, ICWS'09, Los Angeles (CA), USA, July 2009. doi: 10.1109/icws.2009.110.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Service Definition Language (WSDL). W3C, March 2001. URL http://www.w3.org/TR/wsdl.

Massimo Coppola, Yvon Jégou, Brian Matthews, Christine Morin, Lucas Pablo Prieto, Oscar David Sánchez, Erica Y. Yang, and Haiyan Yu. Virtual Organization Support within a Grid-Wide Operating System. *IEEE Internet Computing*, 12(2):20–28, 2008. doi: 10.1109/mic.2008.47.

CORBA. The Common Object Request Broker Architecture. Version 3.1. Object Management Group, January 2008. URL http://www.omg.org/spec/CORBA/3.1.

Francisco Curbera, William A. Nagy, and Sanjiva Weerawarana. Web Services: Why and How. In *Workshop on Object–oriented Web Services*, OOPSLA 2001, Tamba Bay (FL), USA, October 2001.

Karl Czajkowski, Ian Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP'98, Orlando (FL), USA, April 1998. doi: 10.1007/bfb0053981.

Christopher Dabrowski. Reliability in Grid Computing Systems. *Concurrency and Computation: Practice and Experience*, 21(8):923–1108, 2009. doi: 10.1002/cpe.v21:8.

David De Roure, Carole Goble, and Robert Stevens. The Design and Realisation of the myExperiment Virtual Research Environment for Social Sharing of Workflows. *Future Generation Computer Systems*, 25(5):561–567, 2009. doi: 10.1016/j.future.2008.06.010.

Thierry Delaitre, Tamas Kiss, Ariel Goyeneche, Gabor Terstyanszky, Stephen Winter, and Peter Kacsuk. GEMLCA: Running Legacy Code Applications as Grid services. *Journal of Grid Computing*, 3(1):75–90, 2005. doi: 10.1007/s10723-005-9002-8.

Ivo D. Dinov, John D. Van Horn, Kamen M. Lozev, Rico Magsipoc, Petros Petrosyan, Zhizhong Liu, Allan MacKenzie-Graham, Paul Eggert, Douglas S. Parker, and Arthur W. Toga. Efficient, Distributed and Interactive Neuroimaging Data Analysis Using the LONI Pipeline. *Frontiers in Neuroinformatics*, 3(22):1–10, 2009. doi: 10.3389/neuro.11.022.2009.

Guillaume Dugas-Phocion. *Segmentation d'IRM cérébrales multi–séquences et application à la sclérose en plaques*. PhD thesis, École des Mines de Paris, Sophia Antipolis, France, March 2006.

Guillaume Dugas-Phocion, Miguel Angel González Ballester, Malandain Grégoire, Christine Lebrun, and Nicholas Ayache. Improved EM–based Tissue Segmentation and Partial Volume Effect Quantification in Multi–sequence Brain MRI. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, MICCAI 2004, Saint–Malo, France, September 2004a. doi: 10.1007/978-3-540-30135-6_4.

Guillaume Dugas-Phocion, Miguel Angel González Ballester, Christine Lebrun, Stéphane Chanalet, Caroline Bensa, Grégoire Malandain, and Nicholas Ayache. Hierarchical Segmentation of Multiple Sclerosis Lesions in Multi–sequence MRI. In *International Symposium on Biomedical Imaging: From Nano to Macro*, ISBI'04, Arlington (VA), USA, April 2004b. doi: 10.1109/isbi.2004.1398498.

Thomas Erl. *Service–oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River (NJ), USA, 2005.

Alan C. Evans, D. Louis Collins, and Brenda Milner. An MRI–based Stereotactic Brain Atlas from 300 Young Normal Subjects. In *The 22nd Annual Meeting of the Society for Neuroscience*, Anaheim (CA), USA, October 1992.

Rafael Ferreira da Silva, Sorina Camarasu-Pop, Baptiste Grenier, Vanessa Hamar, David Manset, Johan Montagnat, Jérôme Revillard, Javier Rojas Balderrama, Andrei Tsaregorodtsev, and Tristan Glatard. Multi-infrastructure Workflow Execution for Medical Simulation in the Virtual Imaging Platform. In *The Ninth Healthgrid Conference*, HealthGrid 2011, Bristol, UK, June 2011.

Ian Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *Euro–Par 2001*, Manchester, UK, August 2001. doi: 10.1007/3-540-44681-8_1.

Ian Foster. Service-Oriented Science. *Science*, 308(5723):814–817, 2005a. doi: 10.1126/science.1110411.

Ian Foster. Globus Toolkit Version 4: Software for Service–oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, NPC 2005, Beijing, China, November 2005b. doi: 10.1007/11577188_2.

Ian Foster. Globus Toolkit Version 4: Software for Service–oriented Systems. *Journal of Computer Science and Technology*, 21(4):513–520, 2006. doi: 10.1007/s11390-006-0513-y.

Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of High Performance Computing Applications*, 11(2):115–128, 1997. doi: 10.1177/109434209701100205.

Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Global Grid Forum*, GGF4, Toronto ON, Canada, February 2002.

Alban Gaignard and Johan Montagnat. A Distributed Security Policy for Neuroradiological Data Sharing. In *The Seventh Healthgrid Conference*, HealthGrid 2009, Berlin, Germany, June 2009. doi: 10.3233/978-1-60750-027-8-257.

John Geddes, Sharon Lloyd, Andrew Simpson, Martin Rossor, Nick Fox, Derek Hill, Joseph V. Hajnal, Stephen Lawrie, Andrew Mclntosh, Eve Johnstone, Joanna Wardlaw, Dave Perry, Rob Procter, Philip Bath, and Ed Bullmore. NeuroGrid: Using Grid Technology to Advance Neuroscience. In *Symposium on Computer–based Medical Systems*, CBMS 2005, Dublin, Ireland, June 2005. doi: 10.1109/cbms.2005.76.

Bernard Gibaud, Farooq Ahmad, Christian Barillot, Franck Michel, Wali Bacem, Bénédicte Batrancourt, Michel Dojat, Pascal Girard, Alban Gaigard, Diane Lingrand, Johan Montagnat, Javier Rojas Balderrama, Grégoire Malandian, Xavier Pennec, David Godard, Gilles Kassel, and Mélanie Pélégrini-Issac. A Federated System for Sharing and Reuse of Images and Image Processing Tools in Neuroimaging. In *Computed Assisted Radiology and Surgery*, CARS 2011, Berlin, Germany, June 2011.

Tristan Glatard, David Emsellem, and Johan Montagnat. Generic Web Service Wrapper for Efficient Embedding of Legacy Codes in Service–based Workflows. In *Grid–Enabling Legacy Applications and Supporting End Users Workshop*, GELA'06, Paris, France, June 2006a.

Tristan Glatard, Xavier Pennec, and Johan Montagnat. Performance Evaluation of Grid–enabled Registration Algorithms Using Bronze–standards. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, MICCAI 2006, Copenhagen, Denmark, October 2006b. doi: 10.1007/11866763_19.

Tristan Glatard, Johan Montagnat, Diane Lingrand, and Xavier Pennec. Flexible and Efficient Workflow Deployment of Data–intensive Applications on Grids with MOTEUR. *International Journal of High Performance Computing Applications*, 22(3):347–360, 2008. doi: 10.1177/1094342008096067.

Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). W3C, April 2007. URL http://www.w3.org/TR/soap12-part1.

Shannon Hastings, Scott Oster, Stephen Langella, David Ervin, Tahsin Kurc, and Joel Saltz. Introduce: An Open Source Toolkit for Rapid Development of Strongly Typed Grid Services. *Journal of Grid Computing*, 5 (4):407–427, 2007. doi: 10.1007/s10723-007-9074-8.

Herbert Hellerman. Experimental Personalized Array Translator System. *Communications of the ACM*, 7(7): 433–438, 1964. doi: 10.1145/364520.364573.

Derek L. G. Hill, Philipp G. Batchelor, Mark Holden, and David J. Hawkes. Medical Image Registration. *Physics in Medicine and Biology*, 46(3):R1–R45, 2001. doi: 10.1088/0031-9155/46/3/201.

Eduardo Huedo, Ruben S. Montero, and Ignacio M. Llorente. Evaluating the Reliability of Computational Grids from the End User's Point of View. *Journal of Systems Architecture*, 52(12):727–736, 2006. doi: 10.1016/j.sysarc.2006.04.003.

Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Symposium on Operating Systems Principles*, SOSP'09, Big Sky (MT), USA, 2009. doi: 10.1145/1629575.1629601.

ISO/IEC 19757-2:2008. *Information technology — Document Schema Definition Language (DSDL) — Part 2: Regular–grammar–based validation — RELAX NG*. ISO/IEC 19757-2:2008(E). International Organization for Standardization, December 2008.

Clifford R. Jack, Jr., David S. Knopman, William J. Jagust, Leslie M. Shaw, Paul S. Aisen, Michael W. Weiner, Ronald C. Petersen, and John Q. Trojanowski. Hypothetical Model of Dynamic Biomarkers of the Alzheimer's Pathological Cascade. *The Lancet Neurology*, 9(1):119–128, 2010. doi: 10.1016/S1474-4422(09)70299-6.

Mark Jenkinsonn, Peter Bannister, Michael Brady, and Stephen Smith. Improved Optimization for the Robust and Accurate Linear Registration and Motion Correction of Brain Images. *NeuroImage*, 17(2):825–841, 2002. doi: 10.1006/nimg.2002.1132.

Ardalan Kangarloun, Patrick Eugster, and Dongyan Xu. VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime. In *International conference onf Dependable Systems and Networks*, DSN'09, Lisbon, Portugal, September 2009. doi: 10.1109/DSN.2009.5270298.

Tamas Kiss, Gabor Terstyanszky, Gabor Kecskemeti, Szabolcs Illes, Thierry Delaitre, Stephen Winter, Peter Kacsuk, and Gergely Sipos. Legacy Code Support for Production Grids. In *International Workshop on Grid Computing*, Grid 2005, Seattle (WA), USA, November 2005. doi: 10.1109/grid.2005.1542754.

Guilherme Koslovski, Wai-Leong Yeow, Cédric Westphal, Tram Truong Huu, Johan Montagnat, and Pascale Vicat-Blanc Primet. Reliablility Support in Virtual Infrastructures. In *International Conference on Cloud Computing Technology and Science*, CloudCom 2010, Indianapolis (IN), USA, November 2010. doi: 10.1109/CloudCom.2010.23.

Daniel Kouril and Jim Basney. A Credential Renewal Service for Long–running Jobs. In *International Workshop on Grid Computing*, Grid 2005, Seattle (WA), USA, November 2005. doi: 10.1109/grid.2005.1542725.

Filip Krikava, Philippe Collet, and Mireille Blay-Fornarino. Uniform and Model–driven Engineering of Feedback Control Systems. In *International Conference on Autonomic Computing*, ICAC 2011, Karlsruhe, Germany, June 2011. doi: 10.1145/1998582.1998616.

Sriram Krishnan and Karan Bhatia. SOAs for Scientific Applications: Experiences and Challenges. *Future Generation Computer Systems*, 25(4):466–473, 2009. doi: 10.1016/j.future.2008.09.001.

Sriram Krishnan, Luca Clementi, Jingyuan Ren, Philip Papadopoulos, and Wilfred Li. Design and Evaluation of Opal2: A Toolkit for Scientific Software as a Service. In *World Congress on Services*, Services–I, Los Angeles (CA), USA, July 2009. doi: 10.1109/services-i.2009.52.

Alberto Labarga, Franck Valentin, Mikael Anderson, and Rodrigo Lopez. Web Services at the European Bioinformatics Institute. *Nucleic Acids Research*, 35(2):1–6, 2007. doi: 10.1093/nar/gkm291.

Sébastien Lacour, Christian Pérez, and Thierry Priol. Generic Application Description Model: Toward Automatic Deployment of Applications on Computational Grids. In *International Workshop on Grid Computing*, Grid 2005, Seattle (WA), USA, November 2005. doi: 10.1109/GRID.2005.1542755.

Paul C. Lauterbur. Image Formation by Induced Local Interactions: Examples Employing Nuclear Magnetic Resonance. *Nature*, 242:190–191, 1973. doi: 10.1038/242190a0.

Zhi-Pei Liang and Paul C. Lauterbur. *Principles of Magnetic Resonance Imaging: A Signal Processing Perspective*. Biomedical Engineering. Wiley–IEEE Press, October 1999.

Diane Lingrand, Tristan Glatard, and Johan Montagnat. Modeling the Latency on Production Grids with Respect to the Execution Context. *Parallel Computing*, 35(10-11):493–511, 2009a. doi: 10.1016/j.parco.2009.07.003.

Diane Lingrand, Johan Montagnat, and Tristan Glatard. Modeling User Submission Strategies on Production Grids. In *Symposium on High–Performance Parallel and Distributed Computing*, HPDC'09, Munich, Germany, June 2009b. doi: 10.1145/1551609.1551633.

Marco Lorenzi, Nicholas Ayache, Giovanni B. Frisoni, and Xavier Pennec. 4D Registration of Serial Brain's MR Images: A Robust Measure of Changes Applied to Alzheimer's Disease. In *MICCAI Workshop on Spatio–temporal Image Analysis for Longitudinal and Time–series Image Data*, STIA'10, Beijing, China, September 2010.

Marco Lorenzi, Nicholas Ayache, Giovanni B. Frisoni, and Xavier Pennec. Mapping the Effects of $A\beta_{1-42}$ Levels on the Longitudinal Changes in Healthy Aging: Hierarchical Modeling Based on Stationary Velocity Fields. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, MICCAI 2011, Toronto ON, Canada, September 2011a. doi: 10.1007/978-3-642-23629-7_81.

Marco Lorenzi, Nicholas Ayache, and Xavier Pennec. Schild's Ladder for the Parallel Transport of Deformations in Time Series of Images. In *International Conference on Information Processing in Medical Imaging*, IPMI 2011, Kloster Irsee, Germany, July 2011b. doi: 10.1007/978-3-642-22092-0_38.

Ketan Maheshwari. *Data–intensive Scientific Workflows: Representations of Parallelism and Enactment on Distributed Systems*. PhD thesis, University of Nice–Sophia Antipolis, 2011, February 2011.

Cristian Mateos, Alejandro Zunino, and Marcelo Campo. A Survey on Approaches to Gridification. *Software: Practice and Experience*, 38(5):523–556, 2008. doi: 10.1002/spe.847.

Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble. Taverna, Reloaded. In *International Conference on Scientific and Statistical Database Management*, SSDBM'10, Heidelberg, Germany, June 2010. doi: 10.1007/978-3-642-13818-8_33.

Johan Montagnat. NeuroLOG: Compte–rendu de fin de projet. Technical Report NeuroLOG–L12, CNRS, I3S Laboratory. MODALIS Team, Sophia Antipolis, France, February 2011.

Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay-Fornarino. A Data–driven Workflow Language for Grids Based on Array Programming Principles. In *Workshop on Workflows in Support of Large–scale Science*, WORKS'09, Portland (OR), USA, November 2009. doi: 10.1145/1645164.1645171.

Johan Montagnat, Tristan Glatard, Damien Reimert, Ketan Maheshwari, Eddy Caron, and Frédéric Desprez. Workflow–based Comparison of Two Distributed Computing Infrastructures. In *Workshop on Workflows in Support of Large–scale Science*, WORKS 2010, New Orleans (LA), USA, November 2010. doi: 10.1109/works.2010.5671856.

Luc Moreau, Yong Zhao, Ian Foster, Jens Vöeckler, and Michael Wilde. XDTM: The XML Data Type and Mapping for Specifying Datasets. In *European Grid Conference*, EGC 2005, Amsterdam, The Netherlands, February 2005. doi: 10.1007/11508380_51.

Philippe Mougin and Stéphane Ducasse. OOPAL: Integrating Array Programming in Object–oriented Programming. In *Conference on Object–oriented Programming, Systems, Languages, and Applications*, OOPSLA'03, Anaheim (CA), USA, October 2005. doi: 10.1145/949343.949312.

Susanne G. Mueller, Michael W. Weiner, Leon J. Thal, Ronald C. Petersen, Clifford Jack, William Jagust, John Q. Trojanowski, Arthur W. Toga, and Laurel Beckett. The Alzheimer's Disease Neuroimaging Initiative. *Neuroimaging Clinics of North America*, 15(4):869–877, 2005. doi: 10.1016/j.nic.2005.09.008.

Hidemoto Nakada, Satoshi Matsuoka, Keith Seymour, Jack Dongarra, Craig Lee, and Henri Casanova. *A GridRPC Model and API for End-User Applications*. Open Grid Forum, June 2007.

Steven Newhouse. European Grid Infrastructure — An Integrated Sustainable Pan-European Infrastructure for Researchers in Europe. EGI–InSPIRE, April 2011. URL https://documents.egi.eu/document/201.

Thomas Oinn, Mark Greenwood, Matthew J. Addis, Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Christopher Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006a. doi: 10.1002/cpe.993.

Tom Oinn, Matthew J. Addis, Justin Ferris, Darrent Marvin, MMartin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 17(20):3045–3054, 2004. doi: 10.1093/bioinformatics/bth361.

Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew R. Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Chris Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006b. doi: 10.1002/cpe.v18:10.

Silvia D. Olabarriaga, Tristan Glatard, and Piter T. de Boer. A Virtual Laboratory for Medical Image Analysis. *IEEE Transactions on Information Technology in Biomedicine*, 14(4):979–985, 2010. doi: 10.1109/titb.2010.2046742.

Sébastien Ourselin, Alexis Roche, Sylvain Prima, and Nicholas Ayache. Block Matching: A General Framework to Improve Robustness of Rigid Registration of Medical Images. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, MICCAI 2000, Pittsburgh (PA), USA, October 2000. doi: 10.1007/978-3-540-40899-4_57.

Marco Pagni, Jörg Hau, and Heinz Stockinger. A Multi–protocol Bioinformatics Web Service: Use SOAP, Take a REST or Go with HTML. In *International Symposium on Cluster Computing and the Grid*, CCGRID'2008, Lyon, France, May 2008. doi: 10.1109/ccgrid.2008.28.

Erik Pernod, Jean-Christophe Souplet, Javier Rojas Balderrama, Diane Lingrand, and Xavier Pennec. Multiple Sclerosis Brain MRI Segmentation Workflow Deployment on the EGEE Grid. In *MICCAI–Grid Workshop*, New York (NY), USA, September 2008.

Kassian Plankensteiner. Fine–grained Workflow Interoperability Using an Intermediate Representation. In *Open Grid Forum*, OGF30, Brussels, Belgium, October 2010.

Chris H. Polman, Stephen C. Reingold, Gilles Edan, Massimo Filippi, Hans–Peter Hartung, Ludwig Kappos, Fred D. Lublin, Luanne M. Metz, Henry F. McFarland, Paul W. O'Connor, Magnhild Sandberg–Wollheim, Alan J. Thompson, Brian G. Weinshenker, and Jerry S. Wolinsky. Diagnostic Criteria for Multiple Sclerosis: 2005 Revisions to the "McDonald Criteria". *Annals of Neurology*, 58(6):840–846, 2005. doi: 10.1002/ana.20703.

POSIX.1–2008. *IEEE Standard for Information Technology — Portable Operating System Interface (POSIX) Base Specifications, Issue 7*. IEEE Std 1003.1–2008 (Revision of IEEE Std 1003.1–2004). IEEE and The Open Group, December 2008. doi: 10.1109/ieeestd.2008.4694976.

Sylvain Prima, Nicholas Ayache, Tom Barrick, and Neil Roberts. Maximum Likelihood Estimation of the Bias Field in MR Brain Images: Investigating Different Modelings of the Imaging Process. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, MICCAI 2001, Utrecht, The Netherlands, October 2001. doi: 10.1007/3-540-45468-3_97.

Edward M. Purcell, Henry C. Torrey, and Robert V. Pound. Resonance Absorption by Nuclear Magnetic Moments in a Solid. *Physical Review*, 69(1–2):37–38, 1946. doi: 10.1103/physrev.69.37.

Sylvain Reynaud. *Production Grids in Asia*, chapter Uniform Access to Heterogeneous Grid Infrastructures with JSAGA, pages 185–196. Springer, New York (NY), 2010. doi: 10.1007/978-1-4419-0046-3_15.

RFC 5280. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. The Internet Engineering Task Force (IETF), 2008. URL https://www.ietf.org/rfc/rfc5280.txt.

Charles Rich and Richard W. Walters. Formalizing Reusable Software Components. In *ITT Workshop on Reusability in Programming*, Newport (RI), USA, September 1983.

Basil H. Ridha, Josephine Barnes, Jonathan W. Bartlett, Alison Godbolt, Tracey Pepple, Martin N. Rossor, and Nick C. Fox. Tracking Atrophy Progression in Familial Alzheimer's Disease: A Serial MRI Study. *The Lancet Neurology*, 5(10):828–834, 2006. doi: 10.1016/S1474-4422(06)70550-6.

Javier Rojas Balderrama, Diane Lingrand, Johan Montagnat, Erik Pernod, Jean-Christophe Souplet, and Xavier Pennec. NeuroLog: Neuroscience Application Workflows Execution on the EGEE Grid. In *EGEE Conference*, Istanbul, Turkey, September 2008.

Javier Rojas Balderrama, Johan Montagnat, and Diane Lingrand. jGASW: A Service-Oriented Framework Supporting High Throughput Computing and Non-functional Concerns. In *International Conference on Web Services*, ICWS 2010, Miami (FL), USA, July 2010. doi: 10.1109/icws.2010.59.

Javier Rojas Balderrama, Tram Truong Huu, and Johan Montagnat. A Comprehensive Framework for Scientific Applications Execution on Distributed Computing Infrastructures. In *EGI Technical Forum 2011*, Lyon, France, September 2011.

Javier Rojas Balderrama, Tram Truong Huu, and Montagnat Johan. Scalable and Resilient Workflow Executions on Production Distributed Computing Infrastructures. In *11th International Symposium on Parallel and Distributed Computing*, ISPDC 2012, Munich, Germany, June 2012. doi: 10.1109/ispdc.2012.24.

SCA. Service Component Architecture Specification. Version 1.0. Open SOA, March 2007. URL http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications.

Uwe Schwiegelshohn, Rosa M. Badia, Marian Bubak, Marco Danelutto, Schahram Dustdar, Fabrizio Gagliardi, Alfred Geiger, Ladislav Hluchy, Dieter Kranzlmüller, Erwin Laure, Thierry Priol, Alexander Reinefeld, Michael Resch, Andreas Reuter, Otto Rienhoff, Thomas Rüter, Peter Sloot, Domenico Talia, Klaus Ullmann, Ramin Yahyapour, and Gabriele von Voigt. Perspectives on Grid Computing. *Future Generation Computer Systems*, 26(8):1104–1115, 2010. doi: 10.1016/j.future.2010.05.010.

Alex Sellink and Chris Verhoef. Scaffolding for Software Renovation. In *European Conference on Software Maintenance and Reengineering*, CSMR 2000, Zurich, Switzerland, February 2000. doi: 10.1109/csmr.2000.827324.

Martin Senger, Peter Rice, and Tom Oinn. Soaplab — A Unified Sesame Door to Analysis Tools. In *UK e–Science, All Hands Meeting*, Nottingham, UK, September 2003.

Martin Senger, Peter Rice, Alan Bleasby, Tom Oinn, and Mahmut Uludag. Soaplab2: More Reliable Sesame Door to Bioinformatics Programs. In *Bioinformatics Open Source Conference*, BOSC'08, Toronto ON, Canada, July 2008.

Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *Proceedings of the Third International Workshop on Grid Computing*, GRID 2002, Baltimore (MD), USA, November 2002. doi: 10.1007/3-540-36133-2_25.

David W. Shattuck, Stephanie R. Sandor-Leahy, Kirt A. Schaper, David A. Rottenberg, and Richard M. Leahy. Magnetic Resonance Image Tissue Classification Using a Partial Volume Model. *NeuroImage*, 13 (5):856–876, 2001. doi: 10.1006/nimg.2000.0730.

John G. Sled, Alex P. Zijdenbos, and Alan C. Evans. A Nonparametric Method for Automatic Correction of Intensity Nonuniformity in MRI Data. *IEEE Transactions on Medical Imaging*, 17(1):87–97, 1998. doi: 10.1109/42.668698.

Larry Smarr and Charles E. Catlett. Metacomputig. *Communications of the ACM*, 35(6):45–52, 1992. doi: 10.1145/129888.129890.

Stephen M. Smith. Fast Robust Automated Brain Extraction. *Human Brain Mapping*, 17(3):143––155, 2002. doi: 10.1002/hbm.10062.

Stephen M. Smith, Mark Jenkinson, Mark W. Woolrich, Christian F. Beckmann, Timothy E. J. Behrens, Heidi Johansen-Berg, Peter R. Bannister, Marilena De Luca, Ivana Drobnjak, David E. Flitney, Rami K. Niazy, James Saunders, John Vickers, Yongyue Zhang, Nicola De Stefano, J. Michael Brady, and Paul M. Matthews. Advances in Functional and Structural MR Image Analysis and Implementation as FSL. *NeuroImage*, 2004. doi: 10.1016/j.neuroimage.2004.07.051.

Jean-Christophe Souplet, Christine Lebrun, Nicholas Ayache, and Grégoire Malandain. An Automatic Segmentation of T2–FLAIR Multiple Sclerosis Lesions. In *MICCAI MS Lesion Segmentation Challenge*, New York (NY), USA, September 2008.

Jean-Christophe Souplet, Christine Lebrun, Nicholas Ayache, Stéphane Chanalet, and Grégoire Malandain. Revue des approches de segmentation des lésions de sclérose en plaques dans les séquences conventionnelles IRM. *Revue Neurologique*, 165(1):7–14, 2009. doi: 10.1016/j.neurol.2008.04.009.

Jacek Sroka, Jan Hidders, Paolo Missier, and Carole Goble. A Formal Semantics for the Taverna 2 Workflow Model. *Journal of Computer and System Sciences*, 76(6):490–508, 2009. doi: 10.1016/j.jcss.2009.11.009.

Vijay Subramani, Rajkumar Kettimuthu, Srividya Srinivasan, and Ponnuswamy Sadayappan. Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests. In *Symposium on High–Performance Parallel and Distributed Computing*, HPDC–11, Edimburgh, UK, July 2002. doi: 10.1109/HPDC.2002.1029936.

Yoshio Tanaka, Hiroshi Takemiya, Hidemoto Nakada, and Satoshi Sekiguchi. Design, Implementation and Performance Evaluation of GridRPC Programming Middleware for a Large–Scale Computational Grid. In *Proceedings of the Fifth International Workshop on Grid Computing*, GRID 2004, Pittsburgh (PA), USA, November 2004. doi: 10.1109/GRID.2004.20.

Ian Taylor, Ian Wand, Matthew Shields, and Shalil Majithia. Distributed Computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, 2005. doi: 10.1002/cpe.v17:9.

Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323––356, 2005. doi: 10.1002/cpe.938.

Tram Truong Huu, Guilherme Koslovski, Fabienne Anhalt, Johan Montagnat, and Pascale Vicat-Blanc Primet. Joint Elastic Cloud and Virtual Network Framework for Application Performance–cost Optimization. *Journal of Grid Computing*, 9(1):27–47, 2011. doi: 10.1007/s10723-010-9168-6.

Nicholas J. Tustison, Brian B. Avants, Philip A. Cook, Yuanjie Zheng, Alexander Egan, Paul A. Yushkevich, and James C. Gee. N4ITK: Improved N3 Bias Correction. *IEEE Transactions on Medical Imaging*, 29(6):1310–1320, 2010. doi: 10.1109/tmi.2010.2046908.

UDDI. Universal Description Discovery & Integration. Version 3.0.2. OASIS, October 2004. URL http://uddi.org/pubs/uddi_v3.htm.

Arthur van Hoff, Hadi Partovi, and Tom Thai. The Open Software Description Format, August 1997. URL http://www.w3.org/TR/NOTE-OSD.

Tom Vercauteren, Xavier Pennec, Aymeric Perchant, and Nicholas Ayache. Symmetric Log–domain Diffeomorphic Registration: A Demons–based Approach. In *International Conference on Medical Image Computing and Computer Assisted Intervention*, MICCAI 2008, New York (NY), USA, September 2008. doi: 10.1007/978-3-540-85988-8_90.

WMS–JDL. Job Description Language — Attributes Specification for the gLite Workload Management System. EGEE, November 2010. URL https://edms.cern.ch/document/590869.

WS–BPEL. Web Services Business Process Execution Language Version 2.0. OASIS, April 2007. URL http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

WSRF. Web Services Resource Framework. Version 1.2. OASIS, May 2006. URL http://www.oasis-open.org/committees/wsrf.

Yong Zhao, Mihael Hategan, Ben Clifford, Ian Foster, Gregor von Laszewski, Veronika Nefedova, Ioan Raicu, Tiberiu Stef-Praun, and Michael Wilde. Swift: Fast, Reliable, Loosely Coupled Parallel Computation. In *Congress on Services*, Services 2007, Salt Lake City (UT), USA, July 2007. doi: 10.1109/services.2007.63.

Colophon

This document was typeset using XeLaTeX.
Composed in Mitja Miklavcic's FF Tisa,
José Scaglione & Veronika Burian's Ronnia,
Lucas de Groot's Consolas, and
STI Pub Companies's STIX typefaces.

Gestion du cycle de vie de services déployés sur une infrastructure de calcul distribuée en neuroinformatique

L'intérêt va croissant parmi les communautés scientifiques pour le partage de données et d'applications qui facilitent les recherches et l'établissement de collaborations fructueuses. Les domaines interdisciplinaires tels que les neurosciences nécessitent particulièrement de disposer d'une puissance de calcul suffisante pour l'expérimentation à grande échelle. Malgré les progrès réalisés dans la mise en œuvre de telles infrastructures distribuées, de nombreux défis sur l'interopérabilité et le passage à l'échelle ne sont pas complètement résolus. L'évolution permanente des technologies, la complexité intrinsèque des environnements de production et leur faible fiabilité à l'exécution sont autant de facteurs pénalisants.

Ce travail porte sur la modélisation et l'implantation d'un environnement orienté services qui permet l'exécution d'applications scientifiques sur des infrastructures de calcul distribué, exploitant leur capacité de calcul haut débit. Le modèle comprend une spécification de description d'interfaces en ligne de commande; un pont entre les architectures orientées services et le calcul globalisé; ainsi que l'utilisation efficace de ressources locales et distantes pour le passage à l'échelle. Une implantation de référence est réalisée pour démontrer la faisabilité de cette approche. Sa pertinence et illustrée dans le contexte de deux projets de recherche dirigés par des campagnes expérimentales de grande ampleur réalisées sur des ressources distribuées. L'environnement développé se substitue aux systèmes existants dont les préoccupations se concentrent souvent sur la seule exécution. Il permet la gestion de codes patrimoniaux en tant que services, prenant en compte leur cycle de vie entier. De plus, l'approche orientée services aide à la conception de flux de calcul scientifique qui sont utilisés en tant que moyen flexible pour décrire des applications composées de services multiples.

L'approche proposée est évaluée à la fois qualitativement et quantitativement en utilisant des applications réelles en analyse de neuroimages. Les expériences qualitatives sont basées sur l'optimisation de la spécificité et la sensibilité des outils de segmentation du cerveau utilisés pour traiter des Image par Raisonnance Magnétique de patients atteints de sclérose en plaques. Les expériences quantitative traitent de l'accélération et de la latence mesurées pendant l'exécution d'études longitudinales portant sur la mesure d'atrophie cérébrale chez des patients affectés de la maladie d'Alzheimer.

Services Lifecycle Management Using Distributed Computing Infrastructures in Neuroinformatics

There is an increasing interest among scientific communities for sharing data and applications in order to support research and foster collaborations. Interdisciplinary domains like neurosciences are particularly eager of solutions providing computing power to achieve large–scale experimentation. Despite all progresses made in this regard, several challenges related to interoperability, and scalability of Distributed Computing Infrastructures are not completely resolved though. They face permanent evolution of technologies, complexity associated to the adoption of production environments, and low reliability of these infrastructures at runtime.

This work proposes the modeling and implementation of a service–oriented framework for the execution of scientific applications on Distributed Computing Infrastructures taking advantage of High Throughput Computing facilities. The model includes a specification for description of command–line applications; a bridge to merge service–oriented architectures with Global computing; and the efficient use of local resources and scaling. A reference implementation is proposed to demonstrate the feasibility of the approach. It shows its relevance in the context of two application–driven research projects executing large experiment campaign on distributed resources. The framework is an alternative to existing solutions that are often limited to execution consideration only, as it enables the management of legacy codes as services and takes into account their complete lifecycle. Furthermore, the service–oriented approach helps designing scientific workflows which are used as a flexible and way of describing application composed with multiple services.

The approach proposed is evaluated both qualitatively and quantitatively using concrete applications in the area of neuroimaging analysis. The qualitative experiments are based on the optimization of specificity and sensibility of the brain segmentation tools used in the analysis of Magnetic Resonance Images of patient affected by Multiple Sclerosis. On the other hand, quantitative experiments deal with speedup and latency measured during the execution of longitudinal brain atrophy detection in patients impaired by Alzheimer's disease.