Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki,
Informatyki i Elektroniki

Katedra Informatyki

Metodologia komponentowa do konstruowania i wykonywania aplikacji naukowych wykorzystujących zasoby gridowe

Maciej Malawski

Rozprawa doktorska

Promotor: prof. dr hab. inż. Jacek Kitowski

Kraków, 2008

AGH University of Science and Technology
in Kraków, Poland

Faculty of Electrical Engineering, Automatics,
Computer Science and Electronics

Institute of Computer Science

Component-based Methodology for Programming and Running Scientific Applications on the Grid

Maciej Malawski

PhD Thesis
Computer Science

Supervisor: Prof. Jacek Kitowski

Kraków, 2008

Acknowledgements

The author would like to express his thanks to the supervisor of this thesis – Prof. Jacek Kitowski – for the friendly research atmosphere at the Computer Systems Group of the Institute of Computer Science AGH and for valuable advice which helped shape this thesis in its current form.
Special thanks go to my scientific advisor – Dr. Marian Bubak – for his invaluable help, advice and support in my academic and research work, as well as for his energy, initiative and new ideas which made my scientific work possible.
Support from scientific collaborators and colleagues from partner institutions should be kindly acknowledged, including Prof. Vaidy Sunderam and Dawid Kurzyniec from Emory University in Atlanta for hosting me at their lab and guidance during the initial part of this work; Francoise Baude, Ludovic Henrio, Matthieu Morel and Prof. Denis Caromel from INRIA Sophia-Antipolis for fruitful collaboration as well as Prof. Peter Sloot from the University of Amsterdam for valuable inspiration and advice in the area of computational science.
I also owe thanks to my colleagues at AGH who helped me a lot in my daily academic work: Kasia Rycerz, Bartek Baliś, Renata Słota and Włodek Funika. Special thanks go to Piotrek Nowakowski for polishing my English.
I have had the opportunity to collaborate with several colleagues who were preparing their Master's theses and subsequently continued their research in further projects. Thanks go to Tomek Gubała, Marek Wieczorek, Paweł Jurczyk, Maciek Golenia, Marek Kasztelnik, Daniel Harężlak, Asia Kocot, Iwona Ryszka, Eryk Ciepiela, Tomek Bartyński, Michał Placek, Michał Dyrda and Jasiek Meizner.
Numerous students have contributed smaller or larger batches of effort to MOCCA and its applications: I would like to mention Przemek Pelczar, Tomek Jadczyk, Monika Nawrot, Tomek Sadura, Alina Świderska, Antoni Myłka, Maciej Kaszuba and Norbert Kocik.
This research was partially supported by the European Commission ViroLab project, the CoreGRID network of excellence and the Foundation for Polish Science.

Abstract

Since modern scientific applications are typically developed in a collaborative way by teams using diverse technologies, issues such as composition and integration remain a significant challenge. Additionally, the Grid, which is proposed as an infrastructure for e-Science, induces further problems resulting from the distributed, shared, heterogeneous and unreliable nature of resources it offers. For these reasons, programming and running scientific applications on the Grid remains a highly challenging and relevant problem. In this thesis, the author proposes a methodology which aims to address this issue.
The proposed methodology is based on two principles: the first one is to follow a component-based programming model; the second is to use a flexible technology which allows virtualizing the Grid infrastructure. The author demonstrates how this methodology can be implemented by combining the unique features of the Common Component Architecture (CCA) model together with the H2O resource sharing platform, and how it can be enhanced by a set of methods and tools.
The solutions described in this thesis include high-level composition and deployment using alternative scripting and descriptor-based approaches, support for multiple programming languages using Babel, interoperability with the Grid Component Model, and deployment on existing infrastructures. The MOCCA component framework, developed by the author, serves as a basis for those higher-level solutions, as well as for the applications which are used to validate the proposed methodology. Applications and tests included Monte Carlo simulation of formation of clusters of gold atoms, domain decomposition of cellular automata, data mining experiments in the ViroLab virtual laboratory, as well as a set of synthetic benchmarks designed to verify the proposed solutions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the e-science applications with their characteristic features, and describes the Grid which is now offered to scientists as an infrastructure for performing their computational experiments. With this background, the author defines the problem of programming scientific applications on the Grid and outlines the challenges in building a programming environment to solve it. Consecutively, the main goal and the research objectives of this thesis are formulated.

1.1 E-science applications

Contemporary science is increasingly more reliant on computing: from large-scale computer simulations and modeling to scientific data analysis. In addition to traditional theoretical and experimental research, *computational science* has become an equally important method of getting insight into fundamentals of physics, astronomy, chemistry, biology, meteorology and other disciplines [161]. In life sciences, computational methods, often referred to as *in-silico* experiments, are considered complementary to the methods being used in laboratory settings. Computer simulations have become multidisciplinary and operate on multiple scales, requiring not only more computing power, but also close collaboration between scientists. As a consequence, a new term – *e-science* – is used to describe this new paradigm [171].

Recently, researchers have established a new mode of scientific investigations, called *system-level* science, which is defined as *the integration of diverse sources of knowledge about the constituent parts of a complex system with the goal of obtaining an understanding of the system's properties as a whole* [57]. Such an approach does not focus only on individual physical phenomena, but also on their interactions and interrelations in a complex, physical system. An example of such a system-level approach is the earthquake impact prediction, conducted at the Southern California Earthquake Center [57]. It involves combining models of current stress, the fault system, dynamics of ruptures, propagation of shockwaves and the impact of these waves on the surface and buildings. Another example involves investigations in the framework of the Physiome [90] project, which deals with multiscale modeling

of physiological systems, combining various models on the levels of gene, protein, cell, tissue, organ and organ system. In the area of meteorology one can point to tornado prediction systems, which combine data from various sources (such as high-resolution radars) with real-time simulation and data mining [149]. The information technology supporting such large-scale collaborative applications faces challenges which include *team-oriented* investigation, *sharing* of knowledge, models, software and infrastructure, dealing with *heterogeneous* resources and policies, *dynamicity* and dynamic range of system usage scenarios, and supporting *virtual organizations*.

1.1.1 Examples of e-science applications

Scientific applications which are representative of the problems currently faced by e-science were the subject of investigation in the CrossGrid, K-WfGrid, ViroLab and EUChinaGRID research projects. Direct participation of the author in these projects provided a unique opportunity to acquire deeper insight into the characteristic features and requirements of such applications. A brief overview of these applications is included below.

Cardiovascular blood-flow simulation The Lattice-Boltzmann method of blood flow simulation [10] can be used to assist a medical doctor in surgery planing. The simulation is a *parallel* code, which can use *many alternative types* of boundary conditions, collision models etc. The *compute-intensive* simulation engine can be *plugged in* to other pre- and post-processing steps, such as image segmentation, mesh generation and interactive visualization. When developing the application, it is important to frequently switch between different models to examine their behavior under different conditions. Another interesting aspect of this application is the collaboration of the teams of medical doctors, who can remotely participate in the simulation and visualization session [175].

Flood forecasting simulation cascade The simulation of a flood requires calculations of coupled meteorological, hydrological and hydraulical models, which represent the phenomena at different stages and levels of scale [87]. The *high-performance* parallel simulations are then connected into a *workflow* of computations, where the output of a given step serves as an input to the subsequent step.

Simulation of protein folding Prediction of the structure of proteins is a challenging problem in bioinformatics, where many alternative models exist [25]. In a never-born protein study [124], a large number of aminoacid sequences, which are not present in nature, is generated. A sample of this data is then subjected to structure prediction using different models by different groups. Results are then

compared in order to find proteins which may be of interest for experimental investigation. From the programming point of view, the problem can be reduced to running a large number of similar tasks (parameter study), where some of the tasks can be composite (involving several stages), resulting in a more complex workflow. Another challenge is ensuring optimal resource usage with a dynamically changing resource pool. It is also important for scientists to be able to *deploy* their custom-developed software on the computing resources, for conducting alternate computing *experiment* runs.

Simulation of gold cluster formation Clusters of atoms are an interesting form in between isolated atoms or molecules and solid state. Research in this field may therefore be very important for the technology of constructing nanoscale devices. Modeling of clusters involves several energy minimization methods, as well as choosing an empirical potential [186]. These methods are highly compute-intensive, and an optimal result depends on the number of possible iterations and initial configurations for each simulation run.
When using a simulated annealing method, in order to achieve better results, it is necessary to tune such parameters of the model as the function of cooling. This type of model fitting procedure requires, in addition to a *parallel* simulation run, applying an external loop over the model parameters.

Collaborative data mining experiment using Weka in ViroLab The goal of the ViroLab [184] project is to build a *virtual laboratory* for scientists gathering data and conducting *computational experiments* in the field of infectious diseases, focusing on HIV research [160]. A class of relevant experiments involves analysis of patient datasets, in order to discover drug resistance patterns. Data mining techniques can be applied for this purpose, by using existing toolkits such as the Weka library. A virtual laboratory can be then used to execute such *collaborative* scenarios. For instance, one group of researchers can create custom classifier code, which can be then trained using the data provided by another group. Subsequently, yet another group of scientists can use the trained classifier on their own data, to test and validate the quality of predictions.

1.1.2 Summary of the properties of the e-science applications

The model applications, examples of which are described in Sec. 1.1.1, have many common properties. They are compute- and data-intensive, custom-developed by scientists using many programming languages, and used in dynamic scenarios – *experiments* – which involve various levels of coupling and composition types such as parallel or workflow processing. These features can be summarized in the following

way:

Compute- and data-intensive The simulations which are of interest of e-science require large computing power, which is due to increasing the size and complexity of problems which are being solved, as well as the required accuracy of results. Simultaneously, increasing the accuracy of the available instruments, such as physics detectors, medical scanners and meteorological radars leads to generation of huge volumes of data which must be processed.

Used in dynamic scenarios - *experiments* Computational science shares many common practices with traditional laboratory work, which is focused on performing experiments that may form dynamic and complex scenarios. E.g. in the case of iterative model calibration, a number of experiments with different models need to be performed on many types of input data. Following each phase, the parameters of the models are adjusted to find the best fit to the experimental data. Other experiments may involve time-consuming on-line data mining and filtering jobs, which can trigger specific actions if some interesting event is found. This is usually the case when data from scientific instruments is involved, e.g. in the ATLAS detector in high energy physics [79] as well as meteorological applications [149]. Frequently, it is important to monitor the experiment run, and to interactively react to incoming results by changing parameters, rerunning or even redesigning the experiment.

Various levels of coupling and composition types The applications comprising an experiment may involve various levels of coupling: they range from tightly-coupled domain-decomposition simulations which require high-performance computing, through bandwidth-intensive steering and visualization systems, coarse-grained event-based computations to loosely coupled collections of independent parameter-study tasks. Composition models may involve direct connections and synchronization between computing tasks (e.g. in the case of parallel computing and steering), or workflow models where tasks are executed in the order defined by their data and control flow dependencies.

Collaborative Similarly to most other scientific disciplines, e-science applications are collaborative in their nature. This is even more evident in the case of system-level science, where multidisciplinary research is a crucial point. Therefore, the collaborative aspect is not limited to *usage* of applications, but also includes *development*, i.e. planning, design, implementation, testing and integrating different models and application modules.

Written in many programming languages In contrast to enterprise applications, which are often constrained to one major programming language (e.g. Java in J2EE or C# in .NET), e-science applications tend to be more diverse in terms of programming languages used in their development. Most simulations are written in Fortran (usually in Fortran90, but Fortran77 is still used and not just for legacy purposes). For those modules which involve system programming and networking, C plays and important role. Due to the increasing popularity of the object-oriented paradigm, C++ was introduced as a main language for data analysis software in high energy physics. A suitable example is provided by the ROOT framework [23]. For automating the management of experiments and data, scripting languages are frequently used – particularly Python, which is especially useful due to its efficient bindings to native code such as numerical libraries. When network programming and portal-based presentation layers are essential, the Java language tends to become indispensable, and is often supported by dynamic scripting languages, such as Ruby [153]. Such a heterogeneity implies the integration and collaborative development of e-science applications is a real challenge.

Often custom-developed by scientists Despite growing interest and availability of off-the-shelf scientific packages, such as GAUSSIAN [64] or AMBER [35] for computational chemistry, developing custom simulations or analysis code is still a daily task for a large number of scientists across all domains. This issue is especially important and crucial for those who develop new models and simulation algorithms, since expressing them in terms of a computer program is the only way to obtain and validate results. When conducting such research, it is also common practice to develop many versions of a model and many programs respectively, which are then subjected to further investigations in computer experiments.

1.2 Grid as the infrastructure for e-science

Grid infrastructures are now considered the key technological platforms enabling the realization of the e-science paradigm. The term "Grid", which describes the means of providing computational resources to e-science applications, has a very broad meaning and therefore does not yield itself to a single definition. In this section the author refers to the most important attempts do define the Grid, and then provides an overview of the largest Grid initiatives around the world. Here, the focus is on the infrastructure part, leaving more detailed discussions of middleware and programming models for Grids for the state of the art analysis in Chapter 2.

1.2.1 Definitions of the Grid

One of the first definitions which focuses on the user point of view, was produced by Foster and Kesselman in [55] in 1999:

> A computational Grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.

This definition shows the high-level view on the Grid infrastructures, such as the user of e-science applications would like it to be. However, it does not provide much technical insight into *how* such a vision can be realized. The next definition, given by the same authors and soundly expressed in [52], is as follows:

> Grid is a system that:
>
> - coordinates resources that are not subject to centralized control,
> - using standard, open, general-purpose protocols and interfaces
> - to deliver nontrivial qualities of service.

This definition allows one to distinguish non-Grid systems, such as local resource managers or application-specific solutions which do not comply with the proposed checklist. From the scientific applications' point of view, the first item is important, since it emphasizes the fact that the resources on the Grid are shared, which leads to the technical and organizational issues related to accessing them and executing applications thereon. On the other hand, the capability of receiving nontrivial quality of service, such as large computing power or collaborative capabilities, make the Grid attractive and often indispensable in the domain of e-science infrastructures.

IBM proposes another view of Grid computing, which stresses *virtualization* as a very important aspect [93]:

> Grid computing enables the virtualization of distributed computing and data resources such as processing, network bandwidth and storage capacity to create a single system image, granting users and applications seamless access to vast IT capabilities.

Another perspective on the Grid is defined by the authors of the "Physiology of the Grid" paper [56], who proposed the Open Grid Service Architecture (OGSA). The central idea of this concept is to combine the advantages of Grid and Web technologies to create a new paradigm for Grid systems based on services. In this case, the Grid consists of services communicating using standard protocols and complying to specific conventions which facilitate management of virtual organizations and

distributed systems. Service orientation, such as the one expressed in the Service-Oriented Architecture (SOA), provides a high-level virtualization and programming model (see further discussion in Sec. 2.3.1 and 2.2), considered an important step forward from the initial approaches, which focused on providing raw computational power.

The important term *virtualization* is explained as:

> the encapsulation behind a common interface of diverse implementations. Virtualization allows for consistent resource access across multiple heterogeneous platforms with local or remote location transparency. [56].

More recent concepts, expressed e.g. by the Next Generation Grids expert group [136], suggest the enhancement of Grid systems with the capabilities of the Semantic Web [18]. Enriching Grid services with metadata describing the semantics of data and operations allows us to develop and manage applications which support more advanced and demanding scenarios, such as collaborative disaster handling or individual therapy decision support. By exploiting *knowledge* gathered in the metadata, using such tools as ontologies and automatic reasoning, the infrastructure and the applications can become more usable and of higher quality, leading to realization of the service-oriented knowledge utilities (SOKU) paradigm[136].

The aforementioned definitions and concepts of the Grid obviously do not cover all the aspects which are relevant to this broad area of research and development. One important trend can be seen, however, namely the evolution from simple computing (metacomputing) infrastructures which supported batch jobs to more advanced software systems which provide high-level services. Nevertheless, as shown in the following section, most "next generation grids" are currently in their prototype testbed phases, while existing infrastructures are still based on the old-fashioned solutions. Therefore, basic problems such as access to computation, deployment and application management remain a challenge.

1.2.2 Grid infrastructures

In the following paragraphs an overview of the largest Grid infrastructures in Europe, US, Japan and China is presented. An interesting survey and analysis of existing Grid initiatives can be found in the report by Gentzsch [66].

EGEE Enabling Grids for E-SciencE [48] is a European Commission-funded project to provide computing power to European institutions. Its objective is to create a production infrastructure with operation centers and user support. The key user group consists of high-energy physics operating LHC experiments at CERN, but EGEE also supports other scientists from such disciplines as life sciences, geology or

computational chemistry. EGEE started in 2004. The infrastructure currently consists of over 30,000 CPUs and 5PB disk space, and it is expected to grow even larger. It harnesses resources from more than 240 institutions in 45 countries, consisting mostly of loosely-connected PC clusters. The EGEE infrastructure uses both LCG and gLite [49] middleware packages. It provides means to join virtual organizations with access to all resources along with mechanisms for allocation of resources.

Supporting EGEE is important for any realistic programming environment for e-science, since it will give its users access to the largest production Grid infrastructure in Europe.

DEISA The Distributed European Infrastructure for Supercomputing Applications [44] is a European Grid project whose objective is to integrate national high performance computing systems into a production-quality, distributed supercomputing environment. The main focus of DEISA is on integrating high-end computational resources in order to meet the requirements of large, tightly-coupled parallel applications, that should be run as a single platform. The DEISA infrastructure consists of 21,900 processors and its computing power is approximately 200 TFlops. DEISA includes 11 sites from leading European supercomputing centres, accessible using UNICORE [179] middleware.

Grid'5000 project [78] aims at building a highly reconfigurable, controlable and monitorable experimental Grid platform gathering 9 geographically distributed sites in France, featuring a total of 5000 CPUs. In contrast to EGEE or DEISA, focus is not on building a production infrastructure, but rather on creating a large-scale scientific testbed or instrument, enabling conducting experiments in the area of parallel and distributed processing. Currently, Grid'5000 offers nearly 3000 CPUs with 3500 cores utilizing various architectures (XEON, Opteron, Itanium, PowerPC). Grid'5000 provides its own scheduling mechanism for allocating nodes, which can then be directly accessed for installation of middleware and applications being the subject of experiments.

Grid'5000 is particularly well suited as a testbed for scalability and performance experiments with programming environments and applications, which is important in the development process thereof.

TeraGrid is a US project [172] funded by the NSF to create an integrated, persistent computational resource for academic research. Using high-performance network connections, the TeraGrid integrates high-performance computers (TeraFlops), data resources (TeraBytes) and tools, along with high-end experimental facilities around the country. Its role is similar to that of DEISA in Europe. Currently, TeraGrid includes more than 250 TeraFlops of computing capability and more than 30 petabytes

of online and archival data storage. The resources can be accessed using such middleware packages as Condor-G [58], Globus [68] and PBS [143].

Open Science Grid OSG is another large US Grid initiative [142], mainly targeted at high energy physics applications and closely collaborating with EGEE. OSG focuses on high-throughput computing and gathers 130 resources from 30 institutions, offering thousands of CPUs. OSG develops and uses the Virtual data Toolkit (VDT) as its middleware package, based on Condor-G and Globus.

National Grid Service NGS is an example of a national production Grid infrastructure, operating in the UK. NGS provides access to its resources (4 core sites with at least 64 CPUs in project phase 1), using alternative middleware solutions: Globus, GridSAM [178], Condor and gLite.

D-Grid Is a German Grid project, which aims at supporting national e-Science communities, such as HEP, astrophysics, climate research, etc. It gathers 25 core German computing centres and its resources are available using Globus, gLite and UNICORE.

NAREGI is a Japanese national Research Grid Initiative, whose goals are not only to connect major supercomputing centres, but also to develop new middleware, aiming to create a New-generation Supercomputer. The NAREGI middleware, including the Super Scheduler, is based on known concepts of virtual organizations and on standards supported by OGF, such as JSDL and WSRF. NAREGI also develops programming toolkits such as GridRPC and GridMPI [130].

China Grid CNGrid is a project of China's Ministry of Science and Technology, which connects 8 university computing centres and provides the infrastructure for scientific and engineering applications. The project develops its own middleware, called the Grid Operating System (GOS).

1.2.3 Summary of the features of the Grid environments

According to the definitions of the Grid, and based on the specifics of existing infrastructures, it is possible to identify those features of the Grid environment, which are important from the e-science applications' point of view. The Grid infrastructure can clearly meet the increased demand of applications for computing power and facilitate collaboration through sharing of computing and data resources, along with high-speed network connectivity. However, it also possesses some inherent features which make programming and running applications a challenging task:

- The environment is distributed and heterogeneous in terms of computing nodes and also of network links between them (ranging from high-speed inter-cluster connections, through LANs, to international Internet links).
- The computing resources are shared, possibly between different organizations, hence the user does not have full control over them.
- The resources available to the user may dynamically change over time and may be not reliable.
- There is no single Grid middleware package for accessing Grid resources, i.e. a user may have access to many different infrastructures at the same time, each requiring different middleware.
- The infrastructures are based on diverse concepts and programming models.
- Collaborations between the institutions and individuals sharing their resources in virtual organizations can be highly dynamic.

1.3 Problem statement: the need for a methodology

The main problem can be stated as follows: *How to program and run e-science applications on the Grid*? Although significant effort is being invested in research on programming models, tools and environments (see the analysis of the state of the art in Chapter 2), the problem remains challenging and of great importance. Therefore, investigations in this field are still necessary, timely and foreseen as the key research objectives for the community [112, 136]. The answer to this research question will result in a *methodology*, consisting of a set of methods and tools, possibly integrated into a programming environment.

The problem can be expressed in terms of the requirements of the applications and the features of the Grid environment. From the analysis of these features, as summarized in Sections 1.1.2 and 1.2.3, the conclusion can be drawn that there is a need for a methodology, which could facilitate programming and running scientific application on the Grid. The methodology should be supported by an environment characterized by the following features:

- Facilitating high-level programming
- Facilitating deployment on shared resources
- Scalable to diverse environments
- Communication adjusted to various levels of coupling

- Supporting multiple programming languages
- Adapted to the unreliable Grid environment
- Interoperable

As a result, there is need for a *high-level* programming and execution environment based on an appropriate programming model and supported by specific tools and services. The proposed features of the environment aim to make the usage of complex Grid infrastructures simpler and easier for e-scientists.

Below, the desired features are described in more detail.

Facilitating high-level programming The model should allow composing the application from smaller blocks (modules) and express both temporal dependencies between them, as well as direct connections. Combination of both modes (composition in time and in space) is a crucial feature of the model, since both types of interactions are present in the analyzed applications. This composition should be performable by a third party, not hard-coded in the modules. It should also be possible to compose the application on a high level of abstraction, without the need to specify to many technical, infrastructure- and middleware-specific details.

Facilitating deployment on shared resources The environment should support deployment of custom application code on the available resource pool, taking into account the heterogeneity of the infrastructure and middleware. I.e. it should provide a virtualization layer, capable of hiding the diversity of lower layers. The deployment should be dynamic, allowing adaptive application behavior, namely by capabilities of deployment, undeployment and redeployment of code at runtime.

Scalable to diverse environments Due to the heterogeneity of the infrastructure, and also to facilitate application development, the programming environment should be scalable to run on machines ranging from single PCs or laptops, through High Performance Computing (HPC) clusters to multiple Grid sites. In other words, the environment should guarantee that the underlying infrastructure does not determine the programming model.

Communication adjusted to various levels of coupling As the communication layer of the Grid may be very heterogeneous, comprising peer-to-peer networks, WANs, LANs, inter-cluster connections, and even direct binding in a single process, the communication layer of the environment should be able to adjust the connections between application modules to these physical constraints. The communication layer should also support collective or parallel connections between application modules.

Supporting multiple programming languages As the scientific applications may be written in many programming languages, including C, C++, Fortran 77, Fortran 90, Java, Python, etc., the programming environment should not be constrained to only one language. The desired environment should support such scenarios as easy adaptation of legacy code, combining Java flexibility with optimized Fortran libraries.

Adapted to the unreliable Grid environment As the Grid environment may be highly dynamic and undependable, it will be crucial for the environment to provide some means of adaptability and fault tolerance. For this purpose, it should support such monitoring capabilities and adaptive features as dynamic and interactive reconfiguration of connections, locations and bindings, as well as provide support for migration and checkpointing.

Interoperable As it is important for the environment to be usable and not isolated, it should provide mechanisms for interoperability with existing and standard technologies. Some of these standards are defined by the Open Grid Forum, e.g. the OGSA model, which suggests Web services based on SOAP, WSDL, and WSRF as the basis for Grid environments. In the case of specific programing models, it will be important to interoperate with the most popular implementations of these models.

The concept of such an environment and the discussion on how it can be created constitutes the main topic of the author's work presented in this thesis.

1.4 Thesis statement and research objectives

Section 1.3 described the research problem which is being investigated, namely the challenge of programming and running e-science applications on Grid infrastructures. The author also outlined the features which should characterize an environment intended to support this challenging task. Before proceeding to Chapter 2, with the analysis of the state of the art covering the outlined research area, and prior to presenting the concept of the environment in Chapter 3, the author needs to state the main thesis of this work:

> ***Using a component model, enhanced with higher-level programming tools and combined with an appropriate virtualization layer, constitutes a methodology effectively supporting programming and running complex scientific applications on the Grid.***

To validate the proposed thesis, the author proposes the following research objectives:

1. Methodology and concept of a programming environment for scientific applications on the Grid
2. Analysis of programming models for Grid applications
3. Identification of desired features of the programming environment
4. Prototype implementation and feasibility study
5. Verification of the methods and tools with typical applications

Developing a methodology for programming and executing scientific applications on the Grid requires facing challenges, which can be divided into two groups. The first one is the selection of a *programming model* appropriate for the application requirements as well as for the Grid environment. Such a model should be suitable for a distributed environment, enabling management of complex scientific applications. Moreover, it should be supported by standards and encourage good software engineering practices. Subsequently, there is a need to provide a *virtualization layer* to handle and hide the specifics of the Grid environment as well as to allow dynamically creating/acquiring pools of resources. The programming model has to be combined with virtualization aspects and enhanced by abstractions and tools which facilitate using (programming) the resulting environment.

1.5 Note on the projects and collaborations

The author of this thesis has actively participated in several research projects related to the area of programming and running e-science applications on the Grid infrastructures, therefore the discussion and results presented here reflect the perspective and experience gained during this work. The main background is provided by collaboration in a research team at the Institute of Computer Science and ACC CYFRONET AGH.

Participation in the technical architecture team of the CrossGrid project has yielded insight into the development of European Grid infrastructures based on DataGrid, LCG and gLite middleware. The architecture of the project had to cope with demands of the interactive scientific applications, which contributed to the analysis presented in this thesis. Following the evolution of Grid systems towards the service-oriented architecture paradigm resulted in deeper understanding of the impact of programming models on the applications and programming environments.

The author's opportunity to work at the Distributed Computing Laboratory, Emory University, Atlanta, collaborating with developers of the H2O platform had a major impact on the concept of the programming environment, which, combined with collaboration with CCA Forum members, resulted in the main contribution to the proposed methodology. Multilanguage interoperability was developed in collaboration with the Babel team from Lawrence Livermore National Laboratory.

Participation in the K-WfGrid project influenced the concept of a high-level programming model based on multiple levels of abstractions. Participation in the ViroLab project and leading a task responsible for providing the middleware for a Grid-based virtual laboratory enabled deeper investigation and validation of the applicability of a high-level scripting approach for component composition, as proposed in this thesis. At the same time, collaboration within the CoreGRID network helped gain up-to-date knowledge and experience with other component-based and related approaches for high-level programming on the Grid. The interoperability study of GCM and CCA was the result of of the author's CoreGRID research exchange programme fellowship at INRIA in Sophia-Antipolis.

Information on collaborating colleagues is provided in footnotes where appropriate.

1.6 Thesis contribution

The work presented in this thesis, although based on a wide range of collaborative research projects, required substantial personal engagement and contribution of the author.

The main contribution of the author is the component-based methodology, as described in this thesis, including the various elements:

- The author proposes the **concepts** of high-level scripting and ADL-based approaches to support component composition, an underlying base component framework (MOCCA) combining features of CCA and H2O models, an interoperability solution bridging CCA and GCM component models, multilanguage and multiprotocol interoperability for component frameworks by combining RMIX and Babel, and a method of creating pools of component containers and overlay networks to enable deployment of components on production Grid infrastructures and multiple clusters.

- In order to verify the proposed concepts, the author proposed and **designed** the following tools and specific solutions: design of support for CCA components in the high-level GridSpace scripting environment, architecture of the MOCCAccino manager system, design of the MOCCA component-based

framework, design of an interoperability solution bridging CCA and GCM components, design of integration of RMIX and Babel systems.

- The following aspects of the environment were subject to prototype **implementation** prepared by the author: the MOCCA CCA-compliant framework, adapters for MOCCA components for the GridSpace system, implementation of interoperability between GCM and CCA using MOCCA and ProActive, and tools supporting deployment on production Grids.
- To verify the prototypes, the author conducted **experiments** and tests, including specific benchmarks as well as sample scientific applications.

1.7 Organization of the thesis

This thesis is organized as follows: Chapter 2 includes the analysis of the state of the art of the solutions for supporting scientific applications on the Grid, including programming models and such aspects as composition, execution, deployment, interoperability and adaptability. In Chapter 3 the author describes the concepts of a new component-based methodology and outlines the structure of the proposed solutions. In Chapter 4 the concept of a high-level scripting approach to component composition is presented, together with the architecture of the supporting environment. In Chapter 5 the author describes an alternative high-level composition approach based on the Architecture Description Language (ADL) and present the architecture of the proposed manager system. Chapter 6 describes in detail the MOCCA component framework which forms the base of the proposed methods and tools. Chapter 7 presents how the proposed solutions can be made interoperable with other systems, including the Grid Component Model, Babel-based multilanguage CCA components and Web services. Chapter 8 deals with issues of deployment of components on production Grid infrastructures such as CrossGrid and EGEE, and proposes how the JXTA peer-to-peer framework can be used to provide communication with resources in private networks. In Chapter 9 the author includes a description of the applications and tests which were performed to validate the proposed solutions. Finally, Chapter 10 includes a summary, conclusions and future work.

Chapter 2

Scientific Applications on the Grid

This chapter provides an analysis of the state of the art of programming and execution of scientific applications on the Grid. The survey is organized according to the problems related to the main stages of application lifecycle, including programming in a selected model, deployment on environments of varying scales, providing communication mechanisms, adapting to the changes in the environment and interoperating with other technologies.

2.1 Scientific applications - programming and running on the Grid

In order to organize the state-of-the-art survey, the author decided to first describe the process of application *programming and execution*, and to outline the most important issues which have to be discussed. The analysis is intentionally focused on these aspects, since including additional stages, such as experiment preparation, input-output management, result analysis, etc., would overly broaden the scope of discussion. The main steps of the application lifecycle and its requirements are schematically depicted in Fig. 2.1.

The first stage is application programming, where the required functionality has to be encoded in a programming language with the use of an appropriate programming model. A programming model can be defined as:

> an abstract conceptual view of the structure and operation of a computing system [77].

When considering the Grid as a computing system, the programming model has to take into account the structure of the Grid, and define the basic application blocks running on computing nodes, their interactions and the mechanisms of high-level composition.

Due to the fact that the computing resources on the Grid are distributed and shared, once the application is programmed, it cannot be immediately executed.

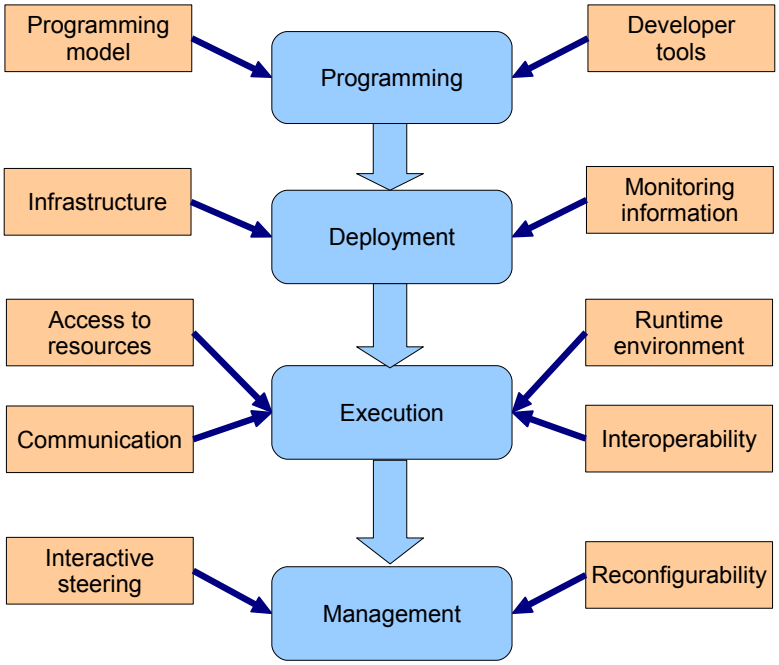


Figure 2.1: Main stages of application programming and execution, along with their requirements

This needs to be preceded by a process which includes installation, configuration, planning, preparation, and launch of the application, shortly named *deployment* [140]. Deployment is particularly important in the case of e-science applications which may be custom-made by scientists and go beyond pre-installed modules.

Once the application code is deployed, the execution of its modules can start. Due to the heterogeneity of the Grid, many possible protocols and middleware systems may exist to enable application execution. Moreover, it may be necessary to use such multiple middleware systems in one, large-scale application. What is interesting is that different visions of the Grid (see Sec. 1.2.1) assume different approaches to application execution: from job submission to invocation of a service.

When the application is running, it may involve interaction with many internal and external modules, subsystems, services, etc. This is particularly interesting in collaborative scenarios, when e.g. a simulation module can be dynamically connected to the visualization or steering system. For such interactions to be possible, an application should provide interoperability mechanisms, possibly by using standard protocols and agreed-upon interaction models.

A long-running application in the Grid infrastructure should also be aware of the dynamic nature of the environment. Node or network failures and changes in the availability of resources may lead to application crash or performance decrease, as well as inefficient resource utilization. Therefore, the need for application management and fault tolerance becomes an important issue.

2.2 Programming models and environments for scientific applications on the Grid

The programming models for the Grid reflect the structure of the environment, so they first have to define how to map the units of the program onto the computing nodes on the Grid, then how to organize communication within the system and finally how to to model a high-level view on the structure of the application. The structure of the Grid can be seen as an extension and combination of parallel and distributed systems, so the natural way of devising programming models is to extend the models existing for such environments. As observed by Laforenza in [112], the complexity of the Grid environment is even higher than that of earlier parallel and distributed systems, making the application programmer's work a more difficult and time-consuming task. Since no single programming model fits all application scenarios, research in the area of models and supporting environments becomes very important.

An analysis of programming models for Grid applications can never be complete, therefore here it will be constrained to covering the most popular and important ones. As a good historical but still up-to-date overview of types of Grid applications the author refers to the paper by Laforenza [112], which should be complimented by the paper by Gannon [63], which, in addition to a technological overview, gives a good classification of Grid users. Here, the discussion begins with an overview of the low-level programming models which define how to construct an application from small units, and later the outline of the methods of high-level application composition is presented.

2.2.1 Basic programming models

This section describes the models, which can be used to map computations performed by a program onto the distributed nodes of the Grid. These models come from parallel and distributed computing, and include task processing, message passing, remote procedure calls, distributed objects, tuple spaces, and component- or service-oriented models.

Job processing

One of the most low-level programming models reflects the technical foundations of the Grid infrastructure, as implemented in most of the existing Grid projects (EGEE, DEISA, etc - see Sec 1.2.2). They assume that the Grid is a collection of computing clusters or supercomputers, each equipped with a batch-processing system such as e.g. PBS [143], and possibly supported by a metascheduler of resource broker managing the whole system.

The single unit of computation is a job (task) which can be submitted to the system together with all required input parameters, environment variables and files. The result of the task processing is usually a set of output files which can be returned to the user or uploaded to a specific Grid storage element. The job processing model usually does not offer any mechanism of communication between tasks, and often the cluster nodes do not provide network connectivity. For that reason tasks can be considered independent and not synchronized at the task level.

Job processing is usually the only model supported on current infrastructures such as EGEE. It requires application developers to use many low-level techniques such as scripting and system tools to build and run their applications. Examples include writing JDL scripts which submit a shell script as a batch job, which in turn uses SSH to launch a process on the head node of the Grid cluster to serve as a proxy for communication [145]; or submitting a shell script which queries the LCG File Catalog (LFC), retrieves a TAR archive from a Storage Element (SE)) using GridFTP, unpacks the archive, runs another script launching the computation, stores the output on the SE and registers it in the LFC catalog [124]. These real-life applications exemplify that a high-level programming model is necessary.

Message passing

The message passing programming paradigm gained popularity and rich tool support in the area of parallel processing, especially for distributed memory machines. The model assumes that there are multiple processes running in parallel and communicating by sending messages, either point-to-point or in a collective manner. The main technologies supporting this model are PVM [166] and MPI [132], implemented as the PVM-3 library as well as MPICH, MPICH-2 and others for computing clusters.

The most widely known implementation of MPI for the Grid is MPICH-G2, which is based on the Globus Toolkit. In addition to support for security and co-allocation of resources compatible with Globus-based Grids, it provides some optimizations of communication to the network topology based on multilevel process clustering [101]. Other solutions focus on deployment of application code and providing connectivity between cluster nodes hidden behind firewalls [91].

The main drawback of MPI is its almost static model of processes, which prohibits adding or removing processes dynamically from the running application. Although the necessary extensions have been proposed in the MPI-2 standard, they have not been implemented in MPICH-G2.

Another implementation, which also covers the MPI-2 standard, is Open-MPI [60]. This system evolved from the LAM-MPI known for Linux clusters, and it is built using a low-level component technology, which allows handling heterogeneous networking protocols in a flexible way. Although the dynamic process model of MPI-2 and good support for heterogeneous networks make OpenMPI better suited for Grid applications, the lack of support for application deployment in the programming model and no mechanisms for high-level composition remain drawbacks of MPI.

Tuple spaces

The idea of tuple spaces, where parallel processes can publish and read data (tuples) was introduced in the Linda [21] programming language. The tuple space offers a programmer a convenient abstraction of shared memory for communication exchange between distributed processes. One example of applying the tuple space approach to distributed computing is JavaSpaces [125] which offers a Java API for accessing a shared space of Java objects. There are also commercial implementations, such as GigaSpaces.com.

Another example of a programming model which can be derived from the tuple space idea is the High Level Architecture (HLA) [86], designed to support distributed simulations. The computational units, called federates, can form larger federations and communicate using the publish/subscribe model of distributed objects. Substantial work has been done to support running HLA applications on the Grid environment, including deployment, monitoring and checkpointing [154, 155].

Although the tuple space concept provides a convenient high-level application view and a mechanism for process coordination, its implementations are limited in terms of performance when scaled to a large number of processes. Therefore the tuple space should be used rather as a high-level coordination mechanism, not as a replacement of direct low-level communication.

Distributed objects

Early approaches to client-server interactions in distributed systems have led to the development of the Remote Procedure Call (RPC) concept. It is based on the idea of invoking procedures between distributed processes as if they were local. This idea evolved into object-oriented programming, leading to a distributed object model and the Remote Method Invocation (RMI) paradigm. The model assumes that the

basic building blocks are objects which communicate by sending messages (invoking methods). When the objects are in different processes, they can communicate using exactly the same methods, provided the support of the middleware responsible for hiding the networking layer. Examples of distributed object technologies include Java RMI, CORBA [138] and DCOM [42].

An interesting approach to distributed object programming is represented by the ProActive library [11], which is based on the idea of active objects. Such objects have only a single thread and communicate using asynchronous method invocations, which return *future* objects. By imposing such constraints on the model, such features as object suspension, checkpointing and migration can be achieved far more easily than in the general case. Other extensions to the model include parallel and group communication, also present in the Ibis project [182].

The object-oriented approach provides an elegant programming model for distributed applications, by providing a useful abstraction of objects and their communication. The main problem with distributed systems such as CORBA is the tight coupling between objects in terms of dependencies, which becomes an obstacle for adaptability and flexibility of applications. To overcome these limitations, more loosely coupled programming models, such as components and services, were proposed.

Software components

The underpinning of component-based systems is the composition of applications from software units with specified interfaces and dependencies. The components can be deployed independently and can be composed by a third party [168]. The component model emphasizes such practices as improvement of software modularity, reuse and the possibility of assembling the applications from independently developed modules. It also facilitates system management by providing mechanisms for dynamic application reconfiguration and adaptation by reconnecting the components at runtime. The component model also supports the separation of concerns paradigm, where such aspects as remote communication, security or transactions can be shifted from the component to the container (framework). Component developers can thus focus on implementing the core functionality, not additional supporting code.

Component-based systems have gained popularity in the industry, leading to standards such as Enterprise Java Beans (EJB) [50], CORBA Component Model (CCM) [139] and DCOM [42]. The scientific community has also expressed its interest in component models. There are examples of adapting CORBA to scientific applications [111], and recently the Fractal model [24] has been used as a foundation for the Grid Component Model (GCM) [40] developed within the CoreGRID project. A good overview of component technologies for Grid systems can be found in [67].

The *Common Component Architecture* (CCA) [36] aims to adapt the component model for high-performance scientific computations.

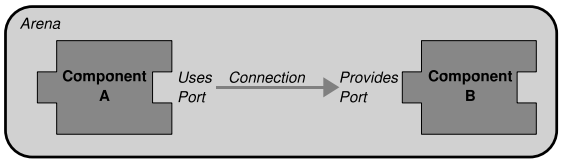


Figure 2.2: CCA component model

Since the subject of author's research is programming of *scientific applications*, the CCA model will be described in more detail. In CCA each component interacts with others through *provides* and *uses* ports. Each component implements a mandatory interface for obtaining a reference to the framework service, which in turn can be used to register ports that the component uses and provides. The process of composing applications in the CCA model has the following form: first, the components need to be created (instantiated) and the *uses* ports of component instances need to be connected to the *provides* ports. The (virtual) container, where component instances are created, is often called the arena, which can be visualized by a GUI tool. In addition, the CCA specification defines a set of standard ports e.g. the *BuilderService* port to create and connect components. It is worth noting that CCA allows components to be dynamically connected and disconnected at runtime. Component interfaces (ports) are described using SIDL (Scientific Interface Definition Language) [104], which is different from other IDLs in that it supports such data types as complex numbers and multidimensional arrays, which are important from the scientific applications' point of view. The CCA does not specify if the components are to be local or distributed, thus enabling both local and distributed frameworks.

One very important aspect of the component-based software model is separation of the interface definition from actual implementation of the component itself. This allows writing components in multiple programming languages and connecting multilanguage components at runtime in one application. This problem is addressed in CCA by relying on Babel [104], which is a SIDL parser and code generator. It is coupled with a runtime library enabling the generation of bi-directional bindings and facilitating interoperability between C, C++, Fortran (77/90), Python and Java.

As the CCA specification matures, many implementations of the CCA standard, called frameworks, are being developed by the scientific community. Each of these frameworks tries to address some of the aspects of the component model, however to the best of author's knowledge none of them simultaneously offer as much flexibility and efficiency as is required by the Grid environment. CCAFFEINE [4] is a framework based on Babel and supporting parallel components based on MPI. It has a

scripting language to compose applications, as well as a GUI control tool. The components in CCAFFEINE are created within a single process, thus inter-component communication is implemented as a local method call. XCAT [106] is a Java-based distributed framework, where components use the SOAP protocol to communicate and it aims at providing WSRF [190] compatibility. XCAT can use ssh or Globus GRAM to instantiate remote components, making it better suited for distributed environments than for high-performance applications. XCAT uses Jython [100] as a scripting language to assemble applications from components. DCA [19] is a parallel framework based on MPI and its main focus is on the MxN problem, i.e. connecting components consisting of M parallel processes to components containing N processes. LegionCCA [73] is a project whose aim is to create a distributed CCA framework for a Legion metacomputing system, however no details are available yet. Scirun2 [192] is a project related to CCA that includes a problem-solving environment with a GUI, designed to support multiple component models. DG-ADAJ is a programming and runtime environment for Desktop Grids [141], which also uses CCA as the basic programming model [6]. CCA is also employed in the Vienna Grid Environment (VGE) [157] as a model for composition of application modules with Web service bindings.

Web services

Following the success of the Web and of the underlying technologies, such as the HTTP protocol and the HTML markup language, the concept of Web services (WS) emerged into the area of distributed systems programming. This model, also known as Service Oriented Architecture (SOA), relies on the exchange of documents (invocations) encoded in XML-based SOAP [189] protocol between clients and services. Services are described using the Web Service Description Language (WSDL) [188] which provides all necessary technical information on how to interact with the service. The premise of Web services is to rely on text (XML) protocols to provide operating system and programming language neutrality. The key concept of SOA is the loose coupling between the services, which should implement the stateless interaction mode.

The Web service paradigm is gaining increased popularity in the world of business applications, and more standards belonging to the WS-* collection are being recommended. In a similar way, Web services are entering the world of scientific computations and particularly into the Grid computing world. The turning point was the announcement of Open Grid Services Architecture (OGSA) by Foster and Tuecke in their *Physiology of the Grid* paper [56] and extended in their later work [53]. OGSA declares the use of Web service standards as a foundation for building Grid systems, and also proposes a set of extensions to the Web services model. Its first extension (although soon deprecated) was called the Open Grid Services Infrastructure

(OGSI). A newer specification, called the Web Service Resource Framework (WSRF) defines the mechanisms of interacting with stateful resources using WS-Addressing and other standards.

The Web service standards are implemented in many frameworks, some of them targeting Grid and scientific computing. The most important example is the Globus Toolkit version 4.0 [68], which provides a container for development and running WSRF-compliant services and a set of higher level services for job management (WS-GRAM) and information management (MDS4) based on the WS concept. Interesting examples also come from the Indiana University, including the xSOAP library, the CCA- and WS-compliant XCAT framework, and a custom implementation of WSRF, WS-Notification and WS-Eventing [61, 89].

2.2.2 High-level programming models

Sec. 2.2.1 provided an overview of how the scientific application can be *decomposed* into smaller units which reflect both the modularity of the application and the distributed nature of the environment. On a high level, a programming model defines how the whole application can be *composed* (again) from these basic blocks, to provide the functionality required by the users. First, an outline of two major composition types is presented: composition *in space* and composition *in time*, and then follow the examples of how they can be realized using specific notations, techniques and tools. In addition to the two fundamental composition types, an overview of the approaches to structured and parallel component composition is provided.

Composition in Space

One deals with composition in space when there are many application units running (possibly in parallel) and they need to interact directly with one another. Examples of composition in space include the creation of a topology of parallel MPI processes (e.g. a Cartesian grid or graph) or connection between the uses/provides ports of the components. A schematic example of such a composition is shown in Fig. 2.3, where three components are connected by direct links. Once the *simulation* component is activated and connected, it can communicate directly with its peers by invoking methods on their ports.

Composition in space can be applied to programming models where basic blocks have direct connections between them. It is thus implementable for MPI processes, for components and for dataflow systems such as Kepler [114].

In the case of MPI, the standard defines an API for creating virtual process topologies, such as Cartesian grids and arbitrary graphs. The can be potentially used by an MPI implementation which uses the virtual topology information to map the physical processes onto the parallel machine. Such a mapping can be used

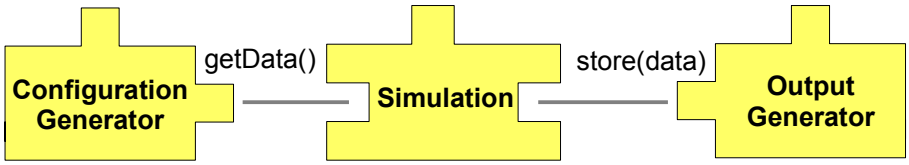


Figure 2.3: Example of using composition in space

to minimize the communication overhead of the whole application and is considered an important research issue [174].

In component-based systems, there are several techniques of composition: low-level API, scripting languages, descriptor-based programming (ADL), skeletons and high-order components, and graphical tools.

Each component standard, such as the Common Component Architecture (CCA) or Fractal, provides an API (possibly in many programming languages) to perform basic operations on components. This API is then used by other high-level interfaces, facilitating the composition process.

One common approach for component composition is to use a scripting language. Some frameworks define their specific notation, as in the case of the CCAFFEINE framework [4], while others offer a direct interface from a script to the framework API. The latter case is realized in XCAT [106], where applications can be assembled using scripts written in Jython [100]. This language has been selected partially because it uses a Java-based interpreter and allows seamless integration with Java client-side libraries or component frameworks. When using such scripts, it is possible to combine composition in time with composition in space, since Python is a powerful programming language which allows expressing the control flow and sophisticated logic of any application.

For component composition in space, it is possible to use an Architecture Description Language (ADL). A good overview of ADLs can be found in [129]. Such a notation, present in the component standards such as Fractal [24] or CORBA Component Model (CCM) [139], allows specifying the application structure in the form of a graph showing connections between components. The ADL approach can be supported by graphical tools, however, it is limited in describing dynamic application behavior and does not allow composition in time.

The Fractal ADL [24], also adapted for the Grid Component Model (GCM), defines the component types, their connections and containment relationships between hierarchical components that allow users to define the component architecture in many aspects related to this component model. Fractal ADL specifies interfaces provided and required by component models, interface cardinality and classes that implement the components. Moreover, containment relationships between components can be declared, attributes can be attached to a component or interface, and

component arguments can be declared. In addition, human-readable comments can be included in the ADL file. By introducing the concept of virtual nodes in ProActive and in GCM [11], it is possible to separate the architecture description from the deployment information, which is then provided in auxiliary deployment descriptor files.

The Grid Application Factory Service [62] for CCA addresses the problem of building reliable, scalable Grid applications. This solution is based on separating the process of deployment and hosting from application execution, and uses XML documents to describe these stages. A new approach to describing component-based application architecture is introduced. It divides information into three XML files. The first file provides basic type information about the component and all the details of its execution environment. This information is necessary and sufficient to launch the component. The second file, i.e. the static application information document, lists components that will be utilized in the computation and connections between them. The dynamic information document, which constitutes the third file, binds every component with its host.

The Corba Component Model (CCM) includes the concept of the IDL3 language, which is responsible for application assembly and deployment. Within the GridCCM project there exists a solution for deploying Corba components on the Grid infrastructure [111] using Globus. ADAGE, which can be used not only for GridCCM, but also for parallel applications, defines its own application description document [111], based on XML.

In the case of Web services, composition in space is an infrequently used paradigm, since there is no standard mechanism for creating direct connections between services. One of the examples, however, is the Grid Service Flow Language (GSFL) [107], which uses a notification mechanism specified in OGSI to directly connect services. Recently, the problem of composition in space has been identified in the SOA model, and a set of specifications called the Service Component Architecture (SCA) [14] has been proposed. The SCA brings into the SOA the old concepts of uses and provides ports, using WSDL to describe the interfaces and allowing implementation of components in many technologies, including C++, Java, Spring or BPEL [137]. It is worth noting that the name of the new standard emphasizes the component nature of the model.

Composition in space can be conveniently supported by graphical tools, which allow building applications by drawing components and connecting their ports with a few mouseclicks. A GUI is offered e.g. by the CCAFFEINE framework, by Fractal and by Kepler, for various actor models supported by the Ptolemy framework. For SCA, there are also graphical tools implemented, some of them based on the Eclipse Platform [47].

Composition in Time - scripts and workflows

Composition in time takes place when there are several tasks which have to be executed in the order of their temporal dependencies, when e.g. the first task produces the output required by the second task. Again, the schematic example of such a composition is shown in Fig. 2.4, where operations on three components are executed in a particular order. The rounded rectangles represent the activities which are connected by the temporal dependencies, representing the control flow of the application. Usually, there is need for some external execution (workflow) engine which triggers the activities and controls the order of execution.

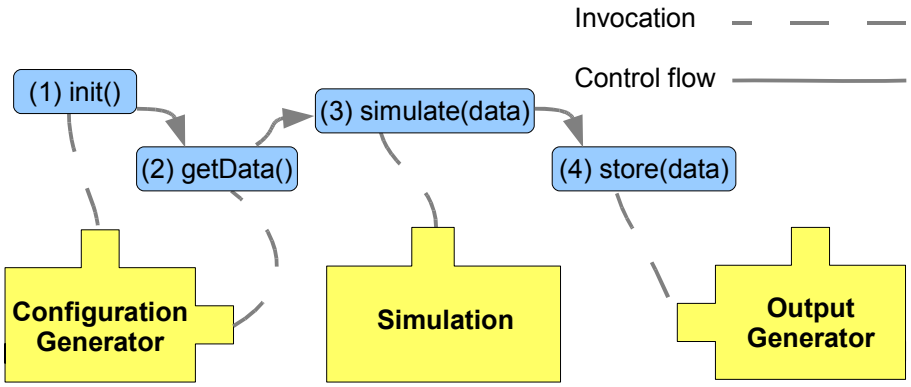


Figure 2.4: Example of using composition in time

Composition in time applies to scenarios where a number of tasks have to be executed in a specific order, determined by their temporal dependencies. Such mode of composition can be applied to the model of computing jobs, as well as to the component and service paradigms.

The basic mechanism for composition in space is to use a low-level API in any imperative programming language. Most of the component frameworks provide client-side libraries in Java or C programming languages. Using them, however, requires good experience with relatively low-level system programming, which is not suitable for non-expert users. On the other hand, the scripting languages [144] are useful for that purpose, since they provide constructs such as pipes and loops which allow expressing the complex control flow of the program. As pointed out in Section 2.2.2, such an approach was successfully exploited within XCAT framework, however the API for interacting with components was quite low-level.

For composition in time, there are more high-level notations available, called workflow languages, which specify application flow (control or data) in the form of a graph. The most widely known example of this notation is based on the concept of representing an application as a Directed Acyclic Graph (DAG) of tasks. It is implemented in Condor DAGMan [173] which reliably manages the execution of

tasks on a pool of Condor-managed resources. Many other workflow systems are also available for the Grid, including Kepler [114], Triana [170], Pegasus [43] and K-WfGrid [82] systems. Workflows can be built from batch jobs and other technologies, such as Web services, possibly combined in one workflow. Systems enable editing the workflow using graphical tools and support specific constructs such as loops, conditions, parallel execution, etc. They are intended to assist non-programmers in developing applications, however in the case of workflows with many components and complex interactions, they can also become difficult to use.

It is also possible to express workflows in a high-level imperative language, such as the ones provided by Grid Superscalar [159] and Grid Application Development Software (GrADS) [17] projects. These systems rely on the concept of a global Grid compilation system, which shifts most of the non-functional decisions from the programmer to an automated systems. The programmer should focus on application functionality (logic) and such issues as parallelization, deployment, management, performance optimization, etc. are ceded to the compiler. As pointed out by Laforenza in [112], the goals of the GrADS projects were ambitious and forward-looking, but to make them real a lot of research in the area of software architectures, base Grid technologies, languages, compilers, tools, environments and libraries is still needed.

Parallel and structured component composition

In addition to composition in space and composition in time, which are of general nature and can be applied to many underlying models (jobs, components, services, etc.), there are also composition techniques which are more specific to the component-based programming model. Here, a description of the parallel composition patterns in component models is given, together with more general, parallel skeleton-based approaches.

One approach to parallelism in component programming is to consider a component as consisting of multiple processes, running on a high-performance parallel machine or cluster. Processes within such a *parallel component* may communicate internally using such mechanisms as e.g. MPI. This approach is being investigated in the CCA [9] model, taking into account such issues as data redistribution for MxN component connections [20]. A parallel extension to the CORBA Component Model (CCM), which also focuses on high performance, is proposed within the GridCCM [148] activity. In this case, parallelism is an aspect orthogonal to composition by connecting interfaces.

Another type of component parallelism can be useful for more distributed and loosely-coupled scenarios, and appears as multiple components of the same type, often referred as *component collections*, running on distributed resources. An example of applying such an approach to the CCA model can be found in the master-worker

application scheme implemented using the XCAT framework [63] or the DG-ADAJ system for Desktop Grids [141]. The ProActive implementation of the Fractal component model [11] offers mechanisms for hierarchical composition of parallel and distributed components, and also defines collective ports based on group communication. These extensions to the Fractal model are included in the Grid Component Model proposal of CoreGRID.

A skeleton-based approach was first proposed by Cole in [38] and is based on the observation that most parallel applications share a common set of composition patterns, such as master-worker, pipe, map or reduce. Therefore a programmer can construct an application by reusing existing application skeletons (which can be optimized and tuned) and inserting only the application-specific functional code into predefined locations. The skeleton approach can also be used to facilitate parallel constructs in component models, and is represented in such projects as skeleton-based ASSIST [1] and HOC-SA [46] introducing the skeleton as a high-order component.

It should be noted that the parallel and structural composition patterns are important for the scientific applications, since they provide the only practical way to exploit a computing infrastructure which is parallel and distributed in its nature.

Composition - discussion

When discussing the possible types of application composition (spatial, temporal, skeleton-based), it must be notes that the choice of the underlying programming model can restrict the high-level composition types available for applications. For instance, the component model can support all types of composition types, whereas e.g. pure service-oriented models do not allow (or no not directly support) composition in space. The results of research in component composition for Grid applications suggests that supporting both composition in space and in time is important. The examples of XCAT [74] and ICENI [128] frameworks suggest that it is possible to combine both types in a single high-level model. In XCAT, however, composition is only possible by using a rather low-level scripting API, whereas in ICENI it requires quite a complex graph-based notation to express the desired application structure. Moreover, recent work by Perez [22] to support composition of GCM components suggests a graph-based notation, which does not necessarily imply a simple solution. Based on experience with the related work, a conclusion can be drawn that supporting component composition on a high level remains an interesting research challenge.

2.3 Grid middleware as the means to provide access to computing resources

Once a scientific application has been programmed, the next important and challenging question can be summarized as follows: *How do I obtain access to computing resources?*.

Addressing this issue requires solutions developed in the form of software platforms which allow aggregation of resources and management of distributed applications. To achieve this goal, the main approach is based on the idea of virtualization of heterogeneous resources, leading to creation of useful abstraction layers. Virtualization can be applied to any layer of the hardware and software stack. This section presents examples of virtualization on the level of batch processing systems, operating systems and software.

2.3.1 Grid toolkits

Among the software platforms which provide virtualized access to computing resources are Grid toolkits, such as the Globus Toolkit [68], Unicore [179] and Legion [134].

Globus, evolving from version 1.0 in late 90s to its current version 4.0.5 (also referred as GT4) provides uniform (thus virtualized) access to various computing systems. These can include remote clusters with local scheduling systems (such as PBS/Torque [143], LSF or Sun Grid Engine – see [191] for a good overview), clusters of workstations managed by Condor [173] or local machines. By using a common protocol called Grid Resource Allocation Management (GRAM and WS-GRAM) it is possible to launch a batch or even an interactive job on the resources which are available. Globus serves as a base middleware stack for the EGEE project [48], as described earlier in Sec. 1.2.2.

Unicore is a solution initially developed by German supercomputing centres, and currently used to provide access to the largest European supercomputing sites participating in the DEISA project [44]. The Unicore approach is very similar to that of Globus, however it uses different protocols and is more focused on providing a user-friendly interface (Unicore client) to non-expert users. There are efforts [150] to extend Unicore to cover such standards as WSRF and JSDL [156].

Legion is a metacomputing project which used to be presented as an alternative to Globus Toolkit. A major difference between both initiatives was that Legion was based on the object-oriented model, tailored to the needs of metacomputing. The Legion object model [113] defines the LegionObject, LegionClass and LegionHosts and their relationships, creation and binding mechanisms, identifiers and security. The core object model is used to provide access to standard programing toolkits

such as MPI and accessing typical applications.

Summarizing, it can be observed that in the case of Globus and Unicore, virtualization is applied on the level of job processing, whereas in Legion it reaches the higher software object level.

2.3.2 Computing access in Service Oriented Architectures

In the case of Service Oriented Architectures, which, since the announcement of OGSA, have become increasingly important and influential for the Grid and e-science systems [53], virtualization is also applied. In Web services all the hardware and software entities are hidden behind a service interface and accessible using a common protocol (e.g. SOAP). In this case, access to computation can be reduced to accessing a specific service. Due to this highest level of virtualization, Web service technologies can provide seamless access to computing resources, however they do not solve deployment problems, described in the following section (2.4).

In addition to using Web services as high-level interfaces to some application-specific functionality (as e.g. in the case of EBI bioinformatic services [110]), it is also possible to use Web services to provide more generic interfaces to computing systems. This is the case of Globus Toolkit 4, where WS-GRAM is a Web service interface for job submission. Similarly, systems such as gLite [49] or Grid-SAM[178] offer WS-based access to EGEE and NGS, respectively. In such cases, we cannot treat these Web services as high-level application interfaces, but rather as internal protocols used by specific types of Grid middleware. Such interfaces are too generic and usually too complex to be directly composed into specific applications.

2.3.3 H2O resource sharing platform

As an alternative to these standard Grid toolkits, the H2O [108] project was initiated, as a lightweight resource sharing platform. The main features of H2O are schematically presented in Fig. 2.5. In H2O, resource providers only need to install an H2O kernel, which serves as a basic container for deploying components, called pluglets. Remarkably important in the H2O model is the separation of the role of container providers (resource owners) from the role of software deployers. This means that a provider can offer a raw resource (CPU, storage) by setting up an H2O kernel with a specified security policy, and other parties (called deployers) can install software by deploying their pluglets into kernels. This deployment process is fully dynamic and facilitated by a simple Java API. Subsequently, the system can be accessed by users, who can utilize the deployed pluglets. The roles of providers, deployers and users can overlap; possibilities are limited only by security policies imposed by the resource owners. This feature makes H2O distinct from OGSI and

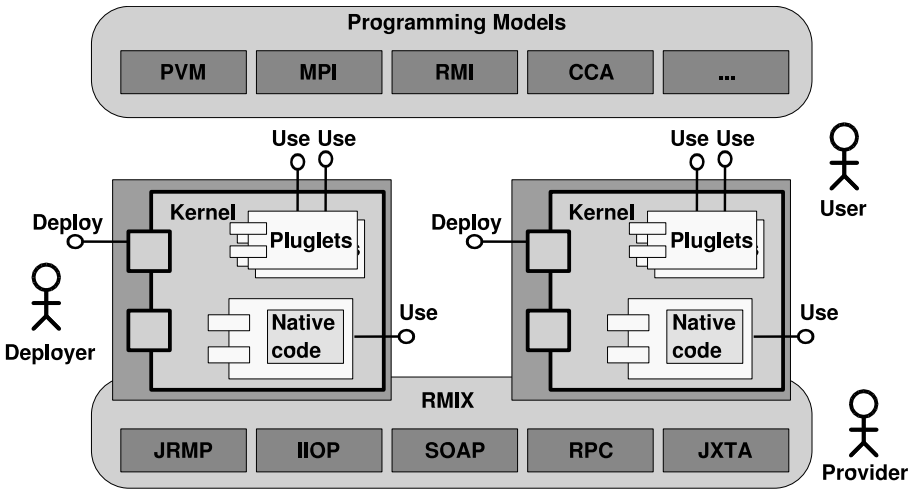


Figure 2.5: Overview of the H2O platform

WSRF, where service deployment is generally bound to resource providers. In addition to this, what makes H2O different from other container-component distributed frameworks, like CORBA or EJB, is the lightweightness and flexibility, where the container provides only basic resource sharing capabilities, leaving all higher-level services to be deployed by clients as required.

It is worth noting that the H2O kernel security mechanisms prevent multiple users from interfering with one another. Each pluglet executes within an access-controlled sandbox, which optionally includes access to a temporary directory created by the kernel in the filesystem of the machine it is running on. H2O API supports staging files to this directory and exchanging data between remote locations, pluglets and client applications. For instance, a user can pass a data file from his machine to the remote pluglet and then return output to an FTP server.

H2O is written in Java and relies on RMIX, an efficient and extensible multiprotocol communication library, based on RMI. It offers an RMI API while allowing protocol negotiation and selection of multiple underlying protocols (e.g. JRMP, SOAP, RPC). RMIX can be used equally well for LAN connections, demanding efficiency, and in widely-distributed environments, focusing on interoperability. Nevertheless, H2O does not mandate the use of RMI as a programming model and Java as a programming language. These features, and additional ones, such as remote file staging, are used as basic building blocks to enable higher-level programming models. Currently implemented models include PVM, MPI [91] and OGSA Web services [164]. H2O can also be used with the JXTA [99] peer-to-peer framework (for details refer to Chapter 8, Section 8.3).

To summarize, in H2O virtualization is applied at the container (kernel) level. Moreover, through usage of Java Virtual machine technology, all lower operating

system and hardware layers can be hidden from the application, thus providing a uniform execution environment.

2.4 Deployment of custom application code

As stated in the introduction, the e-science applications are often custom-developed and can evolve during their development and lifecycle of the scientific experiment. In contrast to local computing facilities, such as clusters with common filesystem, when using Grid infrastructures the process of application deployment becomes a challenge.

According to OMG specification, *deployment* is a process which includes installation, configuration, planning, preparation, and launch of the distributed application [140]. It assumes that the software is developed and packaged and the result of deployment is the start of application execution.

Below the author analyzes the deployment process for:

- Grid toolkits
- Virtual workspaces
- Web services
- Component technologies
- H2O platform

In the case of Grid middleware, such as Globus Toolkit, launching an application is possible by using GRAM or similar mechanisms, as described in Sec. 2.3.1. They provide the low-level means of application installation on the execution host. One option is to use the mechanism of *staging* which means transferring files to and from the execution host, as defined in the JSDL standard, as well as in e.g. the GRAM protocol. Another option is to submit a job in the form of a script, which performs executable transfer (or source compilation) prior to the execution of the application.

Another interesting technology related to application deployment is named Virtual Workspaces [102]. Such dynamically-deployed workspaces can include physical system images of a full operating system installation, as well as virtual machine images or dynamically-created Unix accounts. The Globus Toolkit is used as middleware to provide a service-oriented interface for creating, configuring and accessing the workspaces. As the process of setting up the full system image can require huge data transfers and can be time-consuming, the usage of virtual workspaces only becomes effective for large-scale and time-consuming computations.

An approach similar to virtual workspaces is offered by the *cloud computing* initiatives and the *Platform-as-a-Service* model. Solutions such as the Amazon Elastic Compute Cloud (EC2) [7] allow deploying and running virtual machine images on a configurable infrastructure. Other solutions, such as Google AppEngine [71], provide hosting environments for deploying Web applications developed using a particular technology (such as Python). These solutions, although not suited for scientific applications, demonstrate that the need for software deployment and resource provisioning is important.

The Web services model, while providing good mechanisms for accessing remote services in a loosely-coupled way, does not define any standard mechanisms for service deployment. It is usually assumed that services are deployed by their (and the container's) owner. This issue is noticed by the Grid community which tries to use the Service Oriented Architecture as the basis for building Grid systems for e-Science. There are efforts to provide dynamic service deployment for WSRF containers in GT4, however they are at an early and experimental stage [31].

Component technologies include the deployment process *directly into the programming model*, since a component by definition is the basic unit of deployment. This is an important feature, which makes the component model useful for scientific applications. The deployment is, however, handled differently by different component standards. CCM defines the deployment model very precisely, in its recent version 4 of the *Deployment Specification* [140], however no implementations for the Grid exist yet. On the other hand, there is a research project named ADAGE [111], which proposes tools for automation of deployment of CORBA components on the Grid. ProActive and its implementation of the Fractal component model defines the concept of Virtual Nodes [16], which can be mapped to physical ProActive runtimes. The runtimes (nodes) can be launched using deployment descriptors and can use such mechanisms as SSH, PBS, LSF, Globus, etc., and the components can be subsequently deployed on the nodes using Java dynamic classloading. Recent work on extending the Fractal towards the Grid Component Model also addresses the deployment problem and proposes a common solution for automating this process [39]. CCA does not define any specific standard for deployment, except the BuilderService API, and e.g. the XCAT framework can use SSH and Globus GRAM for instantiating of components, however no installation of software is supported.

The H2O platform, on the other hand, provides a lightweight and simplified deployment mechanism. It uses Java dynamic classloading features, which allow deploying and launching any Java classes published remotely (and possibly packaged as JAR files) on HTTP or FTP servers. The H2O kernel can then dynamically load and launch the code given its location in the form of a URL. As the H2O kernel uses custom classloaders and sandboxes for each of the pluglets, the code, once deployed, can be easily undeployed or redeployed without the need of container restart and

with no interference from other running pluglets.

2.5 Interoperability

Interoperability can be defined as an ability of two or more entities to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction [181]. As stated in the introduction, any realistic environment with ambitions to be usable in the real world has to offer some interoperability mechanisms with other existing solutions. This section focuses on three aspects of interoperability, namely those on the level of the component model, those related to multiple programming languages and those pertaining to multiple communication protocols.

2.5.1 Interoperability between component models

When discussing interoperability between components, the first approach is to rely on existing standards. Unfortunately, there is no single component model or standard defining component interfaces and behavior. The most relevant ones from the scientific and Grid applications' perspective are the CCA, CCM and GCM models and their associated specifications.

Interoperability between components at lower levels, such as the physical, data type and communication level, is not an issue here and will be discussed in the next subsections. Semantic interoperability [84] tries to ensure that requesters and providers in large-scale distributed systems have a common understanding of the meaning of the requested services and data. These aspects have been addressed through design-by-contract and behavioral specification of components. Substantial effort has been devoted to providing semantic descriptions of services using such knowledge management techniques as ontologies [126]; however, this subject is out of the scope of this thesis.

Nowadays, a popular solution for interoperability between components is Web services where standardized protocols based on XML provide a common language for applications to communicate [53]. This approach has been successfully applied to high-performance modules, such as ASSIST modules, wrapped as GRID.it components [3].

In the case of the Grid Component Model, interoperability has been outlined as a requisite: a realistic programming model, in the context of heterogeneous systems, component frameworks and legacy software, must be able to interoperate with existing component frameworks and allow applications to be built from a mixture of various kinds of components. Naturally, the GCM proposes to achieve general component interoperability through standardized Web services.

An interesting approach which has been applied in the SciRun framework [146] is a meta-component model. It allows a single framework supporting components developed in multiple models, e. g. multiple versions of the CCA specification. Such a solution is limited, however, to a single component framework and does not easily allow interactions among multiple frameworks running in parallel.

When discussing inter-component interoperability, one needs to mention standard techniques known from the software engineering integration phase. One of the general benefits of component-based software engineering involves facilitating integration by using composition of components with well-defined interfaces. However, when many different component models exist, integration requires additional steps, such as generation of wrappers, adapters and glue code [162].

2.5.2 Multilanguage interoperability

The existence of many different programming languages causes software interoperability problems, such as data type mappings or language interfaces allowing for native code invocation. To solve the language interoperability problem, several projects have been initiated and some of them are described below.

CORBA CORBA [138] (Common Object Request Broker Architecture) is one of the most popular types of middleware that allows computer applications to work together in a distributed manner. IDL (Interface Definition Language) introduces a way of defining object interfaces that is independent of the programming language (which does not even have to be object-oriented). These interface definitions are used by both server and client sides to enable mutual communication. OMG (Object Management Group), a consortium responsible for CORBA specification development, has standardized mappings from IDL to popular languages, such as C, C++, Java, COBOL, Smalltalk, Ada, Lisp, or Python. As can be seen, the standard lacks a mapping for Fortran, which remains important in scientific environments.

The wire protocol of the CORBA standard is based on the specification of GIOP (General Inter-ORB Protocol) which is abstract and has a standard implementation of IIOP (Internet Inter-ORB Protocol) that provides mappings between GIOP messages and the TCP/IP layer. The CORBA standard requires the protocol implementation's compatibility with the GIOP specs to fulfill interoperability requirements. This makes it a little harder to use many protocols, even if different implementations of GIOP are allowed.

BABEL Another type of interoperability middleware is Babel [104]. The development of Babel was sparked by incompatibilities of many scientific libraries. This libraries are written in languages such as Fortran, C, C++ and others, which, when

assembling compound applications that use many of them, can prove cumbersome. To bypass those difficulties software developers need to create a layer of mediating code and, in the worst case, certain libraries need to be written from scratch or not used at all. Babel helps overcome the issues of interoperability by providing solutions for automatic management of the mediating layer.

The approach is quite similar to CORBA on the interface level. The concept of interface definitions is used, however this time, due to the scientific nature of the problem, SIDL takes the place of IDL, which was used in CORBA. The language has been improved by adding support for complex numbers and multi-dimensional arrays. As for distributed computing, the library has an extension that allows software developers to plug in an implementation of the Babel-RMI library to make Babel objects remotely accessible.

SWIG SWIG [167] is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. The list of supported languages includes scripting languages such as Perl, PHP, Python, Tcl, Ruby and PHP and non-scripting languages such as C#, Common Lisp, Java, Modula-3 and OCAML. Intermediate code generation can be performed one-way only and there is no distributed programming support.

Web services Web services [188] provide a standard means of interoperability between different software applications, running on a variety of platforms and/or frameworks. Web services are characterized by their great interoperability and extensibility, as well as by machine-processable descriptions, thanks to the use of XML. This, however, calls for some additional programming skills on the part of end users, who might just be aware of their area of expertise and have minimal knowledge of XML web APIs. Furthermore, wrapping Fortran code and enabling it as a Web service is also difficult. The protocols used in communication are XML-based, which strongly restricts performance and protocol flexibility.

2.5.3 Solutions for multiprotocol communication

Parallel and distributed computing has gained attention with the advent of Grid computing, peer-to-peer and Web service paradigms [109]. Java-based frameworks in particular have received special attention due to their portability and interoperability. However, the overwide range of applications has imposed many different requirements on wire protocols. Development of more dynamic and loosely coupled applications require runtime protocol negotiation. Many projects try to address these needs and several such frameworks are described below.

RMI Remote Method Invocation (RMI) [165] – an adaptation of the Remote Procedure Call (RPC) for object-oriented environments – is one of the most popular programming paradigms in Java-based distributed systems. The "Standard Edition" Java Platform contains a standard RMI implementation based on the JRMP protocol. The communication libraries are not designed to work cooperatively. For instance, they usually bind clients to a particular wire protocol via stubs that must be generated at compile time, which makes it very inflexible. Many improvements have been made to the standard RMI protocol by projects such as Ibis, KARMI or RMIX, described below.

Ibis [182] is designed as a multi-layer system. On top of the system there are applications, which can use any of the programming models present in Ibis, such as Java RMI (to name just one). Ibis includes an optimized RMI implementation, which can run e.g. on top of Myrinet.

KaRMI [135] is a replacement for the Java remote method invocation package (RMI), which improves the serialization part of the mechanism. The purpose of KaRMI development was adaptation to workstation clusters with high-performance, nonstandard network hardware, making the solution of less general use.

RMIX [109] is a communication framework for Java, based on the Remote Method Invocation (RMI) paradigm. It offers many enhancements over standard RMI. In particular, it supports many wire protocols, which can be developed by third-party providers and plugged during runtime. An object can be remotely enabled through many endpoints, which can in turn be independently configured including using different protocols. The framework also provides end users with asynchronous and one-way calls.

NEXUS Nexus [54] is a portable library that provides multi-threaded communication facilities within heterogeneous parallel and distributed computing environments. It is used in the development of advanced languages and libraries on such platforms. It is not primarily intended as an application-level tool, which makes it difficult to use protocol extensions and further combining it with other software at the application level.

PROTEUS The Proteus [37] multiprotocol library integrates multiple message protocols, such as SOAP and JMS, within one system. It can be seen as a mediator between clients and providers. It uses a different programming model, however, which is based on messaging. Protocol independence is achieved by specifying message contents through so-called *vobjects*. Security is still in the development stage and the protocol switching mechanism is slated for further improvement.

PadicoTM PadicoTM [45] is a runtime system and a framework for integration of multiple communication protocols and paradigms independently of the underlying network. It allows multiple distributed programming environments, such as CORBA, MPI, gSOAP and JXTA to communicate using various networking technologies (Myrinet, Infiniband, SCI, TCP). The goal is high performance and reconfigurability. Although it possesses many interesting features, its disadvantage is the lack of interoperability with Java-based middleware.

2.6 Application management and adaptability

The goal of this section is to provide an overview of the aspects which have to be considered when the application is already running on the computing resources. On such infrastructures as the Grid, the user needs to monitor the progress of the application, as well as to have some means of interacting with it. On the other hand, as the infrastructure may be dynamic an unreliable, there is a need for mechanisms which allow adapting the running application to changes of the environment and provide fault tolerance.

Monitoring of Grid infrastructures and applications is a wide research topic and many corresponding solutions exist. For infrastructure monitoring, systems of interest include Ganglia [127], which is used for monitoring of distributed clusters, MDS [41] which is part of the Globus Toolkit, and JIMS [193], based on the JMX architecture. Application monitoring systems often focus on a specific programming models: examples include OCM-G [12] which was developed for MPI-based applications, GEMINI [177] for workflow monitoring, or TAU [176] performance monitoring for component applications.

Once the monitoring information is obtained, it can be used to configure the application, tune its performance, or perform some other types of adaptation and steering. To make this adaptation process as transparent to the application user as possible, several techniques from the area of autonomic computing [103] can be applied, providing such capabilities as self-configuration, self-optimization, self-healing and self-protection. Examples of applying these techniques to skeleton- and component-based Grid applications can be found in ASSIST [1] and adaptive extensions to GCM [40, 2].

To support the application run in an unreliable environment, additional methods can be applied, including migration and checkpointing. This was the subject of research in e.g. the Dynamite project [94], which supported migration and checkpointing of MPI processes, and G-HLAM [154], supporting HLA applications. In the case of component frameworks, XCAT supports user-level checkpointing [105], while ProActive enables a more transparent approach thanks to the Active Object model [33].

2.7 Analysis of the state of the art

Having described the existing solutions from the perspective of the desired features outlined in Section 1.3, the author summarized the features of the basic programming models (Section 2.2.1) in Table 2.1. The job processing model, although widely supported in Grids, does not offer composition in space and communication between jobs other than via inter-job dependencies. MPI model does not provide high-level composition mechanisms due to rather static application model. Distributed objects lack deployment and composition support in the model, so these important features need to be externally provided. The component model compares favourably to others, since it supports composition *and* deployment directly in the model. Web services, on the other hand, do not support composition in space (direct connections), do not include deployment in the model, and are limited to XML-based communication protocols.

From the analysis, the author can draw the two important conclusions:

- Many of the desired features can be realized if we choose the *component model* as a basic programming model for our environment.
- The desired features of the programming environment are partially fulfilled by the existing environments.

Since the component model looks promising in comparison to other programming models, the author further analyzed the support of the desired features which are present in existing component-based frameworks, targeting either Grid or scientific applications. These features were described in detail in this chapter, and here a concise summary is provided.

Table 2.2 summarizes the features of existing component-based frameworks which are relevant for scientific applications: XCAT, CCAFFEINE, GridCCM, ASSIST, HOC and ProActive, and in last column also Web services are included as an alternative technology. Below, the contents of the table are shortly summarized.

Features \ Models	Jobs	MPI	Objects	Components	Web services
High-level composition	yes	no	no	yes	yes
Facilitated deployment	yes	yes	no	yes	no
Multiple languages	yes	yes	yes	yes	yes
Composition in time	yes	no	no	yes	yes
Composition in space	no	yes	no	yes	no
Flexible communication	no	yes	yes	yes	no

Table 2.1: Comparison of features supported by programming models

- Most component frameworks support composition in space using various techniques, from APIs, through descriptors to skeleton-based solutions. Composition in time was reported as available in XCAT and ProActive. Web services support workflows as their main composition type, although the recent SCA specifications intend to bridge this gap.

- Deployment on shared resources is handled by most Grid-oriented frameworks, where descriptor-based solutions are the most popular ones (ADAGE, GEA, Virtual Nodes in ProActive and GCM). In the case of Web services, deployment is not covered by specifications.

- Target environments range from parallel machines to Grid infrastructures, however it must be noted that the extent of Grid support may vary and is often limited to a single type of middleware (e.g. Globus)

- Most of the frameworks offer a single communication mechanism, using either SOAP or some type of RPC protocol (RMI, CORBA). Support for multiple protocols to adjust communication to various levels of coupling is not present in the frameworks.

- Multilanguage support is not the goal of existing frameworks, with the notable exception of CCAFFEINE, which is based on the Babel tool. Web services also provide multilanguage interoperability, which is a crucial feature for heterogeneous systems integration.

- Most of the frameworks provide some adaptive features, such as dynamic and interactive reconfiguration. Additionally, ProActive and XCAT support migration and checkpointing. On the other hand, ASSIST focuses on autonomic behavior of applications and these features are also being incorporated into GCM and ProActive.

- Regarding interoperability, in most cases it can be achieved by using standard protocols, such as CORBA or SOAP. It should be noted here that ProActive is the only framework which is compliant with the emerging GCM specification.

The conclusion is that although *the component model remains the most appropriate one* for scientific applications on the Grid, *none of the environments fully supports* all the features which have been identified as important (Section 1.3). This conclusion motivates the author to work on new solutions, which is the subject of this thesis.

	XCAT	CCAFFEINE	GridCCM	ASSIST	HOC	ProActive	Web services
Composition in space	Python script	script, GUI	IDL3	skeletons	skeletons	Fractal ADL	N/A
Composition in time	Python script	Low-level API	Low-level API	Low-level API	N/A	workflow	workflow
Deployment specification	script	N/A	XML descriptor ADAGE	XML descriptor GEA	HOC implementation	Descriptor-based	N/A
Deployment technology	Globus, SSH	N/A	Globus, OAR	GEA	Java classloading	Multiple middleware: Globus, local schedulers, SSH, etc.	N/A
Target environments	Grid	Parallel machine	Grid	Grid	Grid	Grid	Web
Communication	SOAP	Local, MPI	Corba, MPI	Corba	SOAP	RMI	SOAP
Language support	Java	Multiple (Babel)	C++	C++	Java	Java	Multiple
Adaptable features	checkpointing	N/A	N/A	adaptive management	N/A	migration, checkpointing	N/A
Interoperability	WS	N/A	CORBA	CORBA, WS	WS	WS	WS

Table 2.2: Summary of the features of component environments for the Grid

Chapter 3

Concept of Component-based Methodology

The author begins this chapter having completed the analysis of the state of the art with the conclusion to select the component model, as it meets most of the relevant requirements. The advantages of selecting the component model are summarized as a rationale for the proposed methodology, which assumes combining CCA and H2O models. This chapter also provides an overview of how the desired features of the proposed new programming environment can be realized when following this methodology.

3.1 Advantages of the component model

When presenting the concept of the proposed component-based methodology, the author begins with answering the following question: why do we consider a component model the most suitable for scientific applications on the Grid? The answer is divided into the following items, which together constitute the rationale behind this choice.

Facilitating high-level programming The component model supports both types of composition: composition in space and composition in time, which allows modelling both workflow and direct communication scenarios, important from the point of view of scientific applications. It is also possible to use semantic descriptions of interfaces to facilitate component composition and interoperability. Additional benefits of the component model are of a more generic software engineering nature: it facilitates code reuse, dependency management and other good practices which are often neglected in scientific programs.

Facilitating deployment on shared resources The concept of a component container and the deployment process are reflected directly in the model. Moreover,

the container provides an abstraction layer which can be used to virtualize the heterogeneous resources available, making it easier to abstract the underlying resources for the application.

Scalable to diverse environments (from laptops to HPC clusters to Grids) The concept of a lightweight container and the mechanism of component composition allow creating applications in a dynamic, pluggable way, thus fitting heterogeneous environments.

Communication adjusted to various levels of coupling By following the separation-of-concerns paradigm, the communication mechanism is provided by the environment, not by components themselves, thus allowing the same components to operate and communicate in both local and distributed configurations, while the protocol layer is managed by the framework. The component models also allows for parallel or group connections and communications.

Supporting multiple languages Many component models facilitate interoperability between components written in various languages, by forcing separation of the interface from the underlying implementation and providing a language-independent interface layer, together with the tools which allow such interoperability.

Adapted to the unreliable Grid environment The component model assumes the possibility of dynamic and interactive reconfiguration of component applications, which makes it especially attractive for long-running computations within a changing environment. By restricting the application to the constraints of a component model, it is also easier to support such features as application migration and checkpointing.

Interoperability Existing standards allow interoperability of component applications, both with Web services and with other component models, such as GCM or CCM.

The author is convinced that the advantages of the component model listed above strongly support its applicability to the problem of programming and running scientific applications on the Grid. The author is also aware of the possible drawbacks of the model, such as tight coupling of components in comparison to e.g. more loosely-connected Web services, or the lack of mature industry standards (CCA, CCM and GCM are mostly of interest to the research community). Nevertheless, the presented advantages of the component model are strong enough to select it as a basis for further research and investigations pursued in the scope of this thesis.

3.2 Concept of the solution – the proposed methodology

Having selected the component model in general, in order to build a programing environment there is a need to focus on a concrete model and choose a base platform for constructing the environment. The selection, which defines the fundamental principles of the proposed methodology, is the following:

- Select the CCA component model.
- Use the H2O platform as a technology.

This decision introduces several benefits, some of which are automatical, and some of which have to be elaborated upon.

To facilitate *high-level programming*, the author proposes to extend the CCA mechanisms by adding the possibility of composition of applications using a *high-level scripting language* or, alternatively, using a manager system based on the *architecture description language*. For *facilitating deployment* the author exploits the dynamic deployment mechanisms offered by the H2O platform and proposes methods for creating a pool of H2O containers dynamically in existing Grid infrastructures, such as EGEE. To support *diverse environments*, it is possible to take advantage of H2O as a lightweight platform and the dynamic reconfiguration capabilities of the CCA model, including changes in component structure at runtime by adding or removing ports. Regarding *communication* adjusted to various levels of coupling, it is natural to rely on the RMIX multiprotocol library offered by H2O and the author proposes extensions for multiple connections between CCA components. CCA gives the possibility to use the Babel system for *multilanguage support* and as a step towards such interoperability, Babel is extended to work with the RMIX communication library. To ensure that applications can be *adapted to the unreliable Grid environment*, the solution is to rely on dynamic reconfiguration mechanisms of CCA and H2O, and to incorporate automatic management features into the high-level programming layers. Finally, to provide interoperability, the author proposes to use Web services but at the same time develops a solution for integrating CCA components with GCM components, implemented in ProActive.

The following subsections outline the concepts of how the desired features will be realized in the programming environment.

3.2.1 Facilitating high-level programming

By relying on the component model, the environment can support both types of composition: composition in space and composition in time. CCA specifies an API

for creating components and connecting their ports, which can be used to provide a low-level composition in space mechanism, by using the Java API or Python and Ruby scripting. On top of it, in order to program on a high level of abstraction and to hide the details of the underlying Grid infrastructure, a high-level scripting layer and an Architecture Description Language-based layer can be built.

A **high-level scripting** layer will be provided to enable application construction using an imperative language. By using a user-friendly API implemented in an object-oriented Ruby script, it will become possible to compose the application on a high level of abstraction, while the underlying runtime system will be responsible for automatic component placement. Additionally, it will be possible to use *the same* Ruby script (referred here as **GScript**) to invoke operations on the created components, using control structures (loops, conditions, iterators, etc.), hence the combined capabilities of both **composition in time and composition in space** can be expressed. The high-level scripting approach is realized as part of the **GridSpace** programming environment, described in Chapter 4.

As an alternative approach, the programming environment will offer an option to specify the application using a declarative language, namely an ADL. It will enable hierarchical composition of component groups, where the actual number of components can be parametrized and dynamically managed. For this purpose, there is need for a manager tool, which will be responsible for automatic deployment of components and management of the application at runtime, reacting to changes in the environment. Such ADL-based composition is realized in the **MOCCAccino** system, described in Chapter 5.

3.2.2 Facilitating deployment on shared resources

By selecting a component model, the problem of application deployment can be reduced to the problem of deployment of components into a container. Having selected H2O as the base platform, we gain access to the H2O kernel, which is a full-fledged application server with remote and dynamic deployment capabilities. Therefore, assuming that there is a pool of H2O kernels available, the underlying Grid infrastructure is *virtualized* as a pool of component containers.

Unfortunately, in current production infrastructures such as EGEE, it cannot be assumed that a pool of containers is automatically available. Therefore there is a need for a mechanism to deploy the kernels using the available Grid middleware prior to actual component deployment. This approach can be seen as dynamic virtualization using a pool of transient H2O kernels created on demand and it is described in detail in Chapter 8.

3.2.3 Scalable to diverse environments

In order to achieve the goal of scalability in environments ranging from single PCs or laptops, through PC clusters to Grids, the environment should be based on two principles: lightweight platform and mechanisms of pluggable and reconfigurable extensions.

As proposed here, H2O can serve as a lightweight platform, since it only requires a Java 1.4 virtual machine, runs out-of-the-box from a 20MB packaged installation, takes ca. 1 sec. to start up on a 2GHz PC and leaves a small memory footprint (approximately 25MB). This makes H2O easy to run on a developer's laptop as well as on a cluster, and easy to deploy on such infrastructures as EGEE.

Regarding reconfigurability, H2O provides hot deployment capabilities, while the CCA model allows for dynamic reconfiguration of component bindings at runtime. Moreover, it is possible to create new component ports at runtime, what may be useful for handling more dynamic scenarios.

The author also reports on the experience with the JXTA framework, which can be used to run component-based application in peer-to-peer environments.

3.2.4 Communication adjusted to various levels of coupling

As introduced in Section 2.3.3, H2O offers a multiprotocol communication library called RMIX for providing remote invocations. Therefore, it can be directly used by components in the following way: components inside a given container can use direct bindings, those located in the same LAN or cluster can use a fast binary protocol, whereas for communication over the Internet, it will be possible to switch on encryption or use the SOAP protocol wherever interoperability is required.

As the computing applications are often parallel, there is a need to introduce some extensions to the model to support parallel connections between components. This is realized by proposing a MultiBuilder extension (see Section 6.4). The parallel connections between component groups are also handled by the MOCCAccino ADL and manager system (Chapter 5).

3.2.5 Supporting multiple languages

Selection of a CCA component model is justified by the possibility to use a solution called Babel, which intends to solve the problem of multilanguage interoperability, including C, C++, Python and Fortran, all important from the scientific applications' point of view.

The proposed approach is to integrate with Babel in two steps: the first one requires integrating Babel with RMIX using the Babel-RMI extensibility mechanism. The concept and prototype of this solution are described in detail in Section 7.3.

The next step will involve integration of Babel with the base component environment, including deployment of components (native code) into H2O. For this purpose, the H2O deployment mechanisms can be used. The concept of this solution is outlined in Chapter 7, but the implementation is left as future work.

3.2.6 Adapted to the unreliable Grid environment

There are several ways to develop a system capable of adapting to such a dynamic environment as the Grid. Dynamic and interactive reconfiguration of connections, locations and bindings is directly supported by the underlying component model (CCA) and by the base platform (H2O).

Some of the automatic adaptive management capabilities are reflected in the design of the MOCCAccino manager system, where it is possible to specify how a system (application) should behave when new containers are added (or removed) from the resource pool. These will be handled by specific annotations in the ADL and by the adaptive behavior of the application manager.

In order to be self-adaptive, a system requires some monitoring capabilities. Our concept assumes two types of monitoring: infrastructure-centric and application-centric. Infrastructure monitoring is realized as part of the GridSpace platform and it provides information to the application optimizer module, which is responsible for managing automatic component placement.

Another issue involves providing fault-tolerance support by mechanisms such as migration and checkpointing. Solutions to these problems already exist in component environments (e.g. in ProActive and XCAT), so it will be possible to incorporate them in the proposed programming environment. These issues, however, are out of the scope of this thesis and are left for future work.

3.2.7 Interoperability

Interoperability of the environment with more standard solutions is crucial for its usability and for its applications. The goal of the proposed concept is interoperability with Web services as a standard for programming distributed systems, and with the Grid Component Model, which is an alternative component model supported by the CoreGRID network of excellence.

By selecting H2O with RMIX, which supports SOAP as one of the protocols, interoperability with Web services is theoretically possible. However, the fact that RMIX does not support WSDL becomes an issue. The author therefore considers using additional Web services layer on top of H2O, so that the provided component ports can be exported as Web services using a modern embedded framework, such as XFire or Apache Axis/CXF.

Since CCA is not the only one component model, and CCM and GCM are also being developed, it becomes important for the presented environment to allow components from one framework to be instantiated in a container provided by another framework and also to allow inter-framework interoperability. In this thesis, the author describe an approach based on the adapter concept, which enables both types of interoperability between CCA and GCM. Therefore, it is demonstrated, that the details differentiating various component models can be, to some extent, hidden, thus exposing another benefit of component models. A detailed discussion of these issues is presented in Chapter 7.

3.3 Structure of the proposed solutions

The component-based methodology, as proposed in this thesis, relies on selecting the CCA model and using the H2O platform to build an environment for programming and running scientific applications on the Grid. This section begins with presenting the structured overview of the concept of the proposed environment and outlines the methodology of using the various elements of this environment for specific purposes. The second subsection introduces important concepts of the base environment which forms the basis for higher layers.

3.3.1 Structure of the concept of the environment

The concept of the environment can be presented as a layered architecture, outlined in Fig. 3.1. On top of the picture, there is the scientific application, developed and executed by users. The applications can be built using any of the lower layers.

Below, there is a high-level composition layer, comprising two composition modes: GScript for the scripting approach and the descriptor-based MOCCAccino system for composition based on the architecture description language. As discussed in Section refsec:composition-space, these modes are alternative approaches, so they can be used depending on the preferences of the developer and on the application type. The GScript approach is better suited for rapid application development, experiments and steering, while MOCCAccino should be used for structured applications which require automated management.

Both modes of composition can use the underlying layer, which includes parallel extensions to the CCA model and techniques for interoperability with other component models. Both are optional, hence if no parallelism or interoperability are required, this layer can be skipped.

The above mentioned layers are built on top of base component frameworks. MOCCA is a component framework implementing the CCA model with the use of the H2O platform. MOCCA can be extended with support for the Babel system,

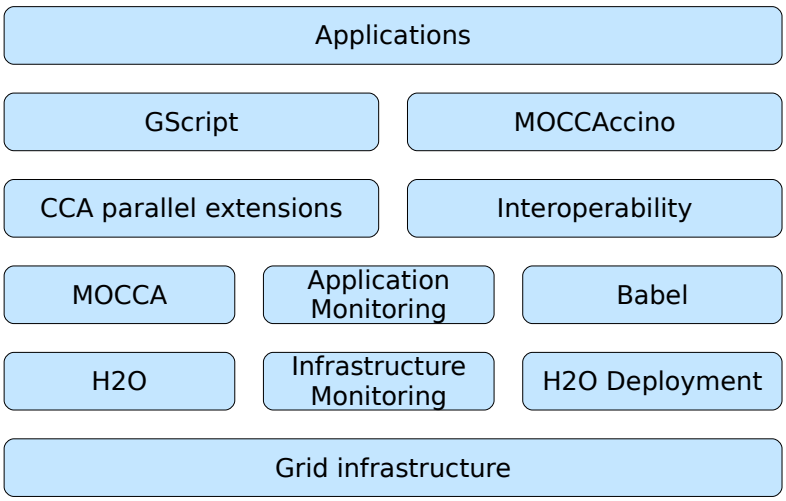


Figure 3.1: Outline of the layered architecture of the proposed environment

providing programming language interoperability. Additionally, application monitoring can be plugged in, supporting application instrumentation and publishing information about application state.

Below are basic middleware technologies, which include H2O as a resource sharing platform and execution environment, infrastructure monitoring providing system status information and techniques for deployment and management of the pool of resources.

The lowest layer is the Grid infrastructure, which, in principle, can include many different middleware types, as the role of higher layers is to hide them from the component model and the application itself.

3.3.2 Basic concepts of the underlying environment

As the resulting component environment will be based on CCA and H2O, in this section the author introduces the most important concepts of the underlying environment, called MOCCA. The vocabulary presented here will be used in subsequent chapters, starting from top the topmost layer.

MOCCA is a Java-based CCA-compliant component framework, implemented in the H2O platform. It provides the basic mechanisms of component creation, connection and execution. **Kernel** is a H2O container, where MOCCA components are deployed and executed. **Component** in MOCCA is a Java class, implementing the CCA `Component` interface. Components are packaged as JAR files. **Deployment**

of MOCCA components is realized using H2O remote classloading mechanisms, i.e. a component JAR package can be dynamically loaded by a container from a remote URL (e.g. HTTP server). Components can be unloaded and reloaded with no need to restart the container. **ComponentID** in MOCCA is a hierarchical URI, constructed from the URI of the H2O kernel and the name of the component pluglet. **Binding (connection)** between MOCCA components is realized using the RMIX library, included in H2O. It allows transparent RMI-style communication between ports of components. **Builder Service** in MOCCA is implemented as a hierarchical system of builder pluglets, with a MainBuilder service, which is a user contact point in the framework. **MOCCA MainBuilder** offers a Java API for interaction with the framework. It also supports scripting using embedded Python and Ruby interpreters.

As can be seen in the short introduction to MOCCA, it is an implementation of the CCA specification which uses the H2O platform and can serve as the basis for supporting higher-level layers. More details of MOCCA, including a discussion on its objectives, design decisions and some implementation details, are presented in Chapter 6.

3.4 Summary

This chapter presented the most important concepts of the proposed component-based methodology. Shortly summarizing, the concept relies on selecting CCA as a component model and on using H2O as the platform on which to build the environment for scientific applications on the Grid. The methodology recommends using the proposed high-level programming tools, depending on the type of application: for rapid development and experimentation a scripting-based environment is offered which allows programming on multiple levels of abstraction, whereas for application of a stable structure a manager system based on an architecture description language is offered. For dealing with interoperability issues, the author proposes solutions for interfacing CCA and GCM components and frameworks. For multilingual support the author suggests Babel, and Web services for interoperation with other external systems. For exploiting the processing capabilities of production Grid infrastructures, the author proposes the methods of deployment of H2O kernels thereon and creating a virtual overlay network using e.g. the JXTA technology.

The concepts presented in this chapter are based on the ideas and experience with existing component-based frameworks for Grid and scientific applications, as discussed in the state of the art analysis. The advantage of the solutions proposed here results from the unique features of CCA and H2O, which, combined together, can offer a new environment, tackling known issues which affect scientific applications on the Grid. The wide range of methods and tools constituting the proposed

methodology and the supporting environment, which covers a broad area of relevant problems, should also be treated as important contribution. It should be emphasized that the proposed features of the higher-level methods and tools such as scripting-based composition, deployment or interoperability are of a generic nature and are not limited to CCA or even to component-based models. The choice of CCA and H2O can be considered as a practical decision, which helps focus and narrow down the research work while at the same time leveraging the benefits of both two solutions.

One of the goals of this chapter was to propose a methodology which would ensure that the related environment is as complete as possible. The author believes that this is the case, although some aspects, such as autonomic behavior, semantic descriptions or security could be addressed more thoroughly. This would, however, dilate the scope of the thesis and therefore some of the interesting research questions had to be left for future work. For similar reasons, the author does not demonstrate a complete and integrated programming and operating environment. Instead, the following chapters will describe those solutions which are considered as being of key importance. Following this approach, the author shows that the component environment based on CCA and H2O can be used to support both high-level scripting and ADL-based composition, interoperability with Babel and GCM as well as deployment on production infrastructures, all of which are described in the following chapters.

Chapter 4

High-level Scripting Approach

This chapter contains a detailed description of a scripting approach for composition of component-based applications. The author argues that incorporating such an approach is very important and valuable for the programming and execution environment being the subject of this dissertation. The goal is to provide a high-level notation combining composition in time and composition in space together with a rich set of control structures and object-oriented abstractions. Following a discussion on the proposed notation, the architecture of an execution engine which is required to support such a programming model is introduced. The chapter also includes a description of optimization strategies in the proposed system.

4.1 Introduction

According to the main concept outlined in Chapter 3, the proposed environment should support a high-level composition mechanism for an application built from components. There are two ways of composing components: composition in space and composition in time (see also Section 2.2.2). Both are relevant to Grid applications [128, 74]. *Composition in space* involves direct connections between component ports, while control and data flow pass directly between connected components. Composition in space can be either *static* – the connections are established prior to application execution, or *dynamic* – the connections may change during application execution which may involve reconfiguration or creation of connections on demand. *Composition in time* assumes that components do not have to be directly connected, but their server interfaces can be invoked by a client, which coordinates the whole application. In this case, both control and data flow pass through the client, which can be a specific application or a more generic workflow engine [82].

There is a need for a high-level programming approach which would enable combining both types of component composition in a way which is flexible and convenient for the programmer. The approach should not be limited to a single component model, since many models are available for programming Grid applications [67]. Moreover, being focused on the Grid environment, it should conceal the

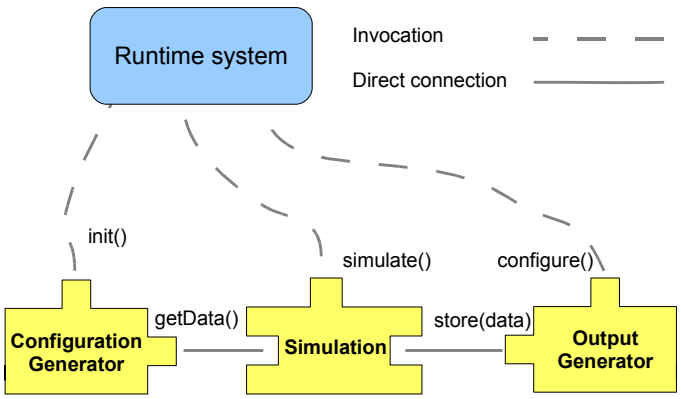
complexity of the underlying infrastructure, automating the process of component deployment and resource selection wherever possible. It would also prove valuable if the solution could facilitate such aspects as component configuration and passing parameters to the application. Such a solution would eventually form a powerful application development, deployment and execution tool for the Grid.

This chapter describes a top-down approach to solving the problem of component composition on the Grid. The proposed solution is based on a dynamic scripting language [169]. This solution is especially well suited for rapid application development (RAD), prototyping and experimenting with scenarios. The scripting approach also provides the full flexibility of using a programming language, enabling a rich set of control constructs for component applications (workflows). The high-level notation can hide all the details of the underlying Grid infrastructure, so the programmer may focus on the application logic while the process of resource selection and component deployment is automated. The proposed approach is in line with the suggestion that any successful component-based platform should support scripting as a mechanism for gluing application components together [144].

The following sections of this chapter present the concept of a scripting language (called GScript)) and show how it can be applied to the Common Component Architecture (CCA) [9] with the MOCCA framework [123] as well as the Grid Component Model (GCM) [65] using the ProActive [11] implementation. After introducing the basic features of the notation, the author briefly describes the architecture of a runtime system (called GridSpace, see also [81]), needed to support the high-level functionality. Finally, a report on the progress of a prototype implementation is presented along with conclusions and prospects for future work.

4.2 Composition with a high-level scripting language

The proposed approach to composition of component-based Grid applications relies on a dynamic scripting language. This basis allows designing a high-level API for application composition and deployment, which enables specifying the application structure in a concise way. Modern scripting languages allow the programmer to specify the same functionality with considerably less lines of code than e.g. Java, making the code more readable and thus less error-prone. Additionally, they provide full expressiveness needed to specify application behavior in a more flexible way than any workflow notation. Following careful analysis of possible candidates, the decision was made to select Ruby [153] which is an object-oriented, dynamic scripting language with a clear and powerful syntax. As an interpreter, JRuby [96] was chosen. This interpreter is implemented in pure Java and allows seamless integration with all




```
generator = GS.create("org.example.ConfigurationGenerator")
simulation = GS.create("org.example.Simulation")
output = GS.create("org.example.OutputGenerator")

simulation.inputPort.connect(generator.dataPort)
simulation.outputPort.connect(output.outputPort)

generator.init(steps, size)

simulator.simulate()
```

Figure 4.1: Sample application using composition in space

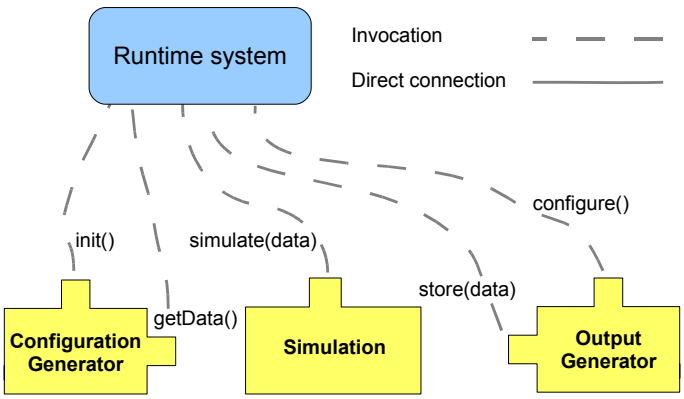
available Java libraries. One of the advantages of choosing JRuby over Jython [100] (Java implementation of Python) was better stability of the available interpreter, important from the development point of view. It should be emphasized, however, that using JRuby is mostly an implementation choice, hence the proposed concepts, solutions and design of the whole system remain independent of the programming language and any specific interpreter.

To illustrate the concept of a script, let us consider a simplified application, consisting of three components (see Fig.4.1 or Fig.4.2):

- *Generator* for preparing initial data,
- *Simulation* part performing some computations and
- *Output* element responsible for storing the results.

Such an application can be modeled either using direct connections between components, or as a workflow which is coordinated by an external entity, labeled as the *RuntimeSystem*.

As Ruby is an object-oriented language, component instances in the script are represented by objects. With dynamic method definition and invocation, it is possible to refer to ports and port operations using a single method. Simple loops




```
generator = GS.create("org.example.ConfigurationGenerator")
simulation = GS.create("org.example.Simulation")
output = GS.create("org.example.OutputGenerator")

generator.init(steps, size)

for i in (1..steps)
    data = generator.getData()
    result = simulator.simulate(data)
    output.store(result)
end
```

Figure 4.2: Example application using composition in time

enable the programmer to create collections of components and then iterate over them or connect them in required topologies, such as graphs or meshes. It is also possible to mix various types of composition and control the dynamic behavior of the application.

4.2.1 Composition support

Script enable the programmer to easily express both types of composition (see Fig. 4.1 and 4.2) while preserving a clear and concise syntax. In Fig. 4.1 the script is used to create direct connections between component ports, but also to configure the components and launch the simulation. Subsequently, control is passed to the components and data flows directly between them, using established bindings (e.g. execution of *simulate()* invokes *getData()* and *storeData()* methods on bound components whenever needed). In Fig. 4.2, the components are implemented in such a way as to not require connections so data is passed through the runtime system. Such composition may be useful for more loosely-coupled scenarios, since it does not require direct dependencies between components.

In both scenarios the *Generator* and *Output* components may be implemented in the same way, hence their usage is invariant in both composition types. This

is, however, not the case for the *Simulation* component, since it needs to get *data* either via a connected *uses* port or as a parameter of the *simulate(data)* method. Nevertheless, it is possible to design and implement a *Simulation* component, which can be compatible with both types of interactions. It should have both *uses* ports and two versions of the *simulate()* method: *simulate()*, which tries to fetch the *data* from the uses port and then store it via the output port, and *result simulate(data)*, which does not rely on the uses port.

4.2.2 Deployment specification

A programmer assembling a Grid application should be free to specify how much information about deployment is to be provided manually and which decisions should be left for automatic tools. Three levels of detail should be considered:

- Fully automatic: the programmer specifies only the class of a component to create. Component deployment location is determined automatically by the system:
  `GS.create(componentClassName)`
- Using a virtual node: the programmer specifies a virtual node on which the component should be deployed:
  `GS.createOnVN(componentClassName, vn)`
- Manual, by specifying a concrete location, e.g.
  `GS.createConcrete(techInfo)` where `techInfo` is the descriptor specifying all concrete information required to create the component (e.g. H2O kernel in the case of MOCCA) and to invoke its methods (e.g. names of the ports).

All these levels should be supported by the runtime system and might be combined by the programmer, e.g. to specify a concrete location for a master component and then let the system automatically select resources for worker components from the available pool.

4.2.3 Framework interoperability

The scripting API for composition and deployment of components is neutral with respect to the component model used. Script invocations are translated into underlying Fractal, CCA or CCM APIs. If more than one component model is supported, it is possible to combine components from different models in a single application. In the case of a component workflow, the runtime system is the central point of inter-component communication, so it acts as an intermediary passing results from one component invocation to another. It is also possible to integrate modules developed in other technologies, e.g. Web services or WSRF, in such a workflow. For

composition in space, when direct links between components are involved, it may be necessary to introduce *glue* ports between the heterogeneous components, to enable translation of invocations between them. As the author's research on interoperability between GCM and CCA [118] suggests, it is possible to introduce such a generic glue which can bridge components from different models and frameworks.

4.2.4 Optimizing communications

As can be seen in Fig. 4.2, composition in time requires all data to flow through the central workflow engine (runtime system), which may lead to bottlenecks and poor scalability in larger systems. One of the solutions to this problem is to use a *pass-by-reference*-like model, where a *data* object does not contain any actual payload, but only a reference to data, e.g. in the form of a URL. This requires both producer and consumer components to support storing and retrieving data from URL-specified locations, but the actual data transfer can proceed directly between them.

There is, however, an alternative solution, which involves *futures*. The statement:

```
data = generator.getData()
```

may not block until the invocation is realized, instead creating a promise (a *future*) for the actual data. Subsequently, invoking:

```
result=simulator.simulate(data)
```

will copy the future reference to the simulator which will automatically retrieve the value of the data element when the generator computes it (and it will be automatically blocked whenever data is needed but has not yet been computed and received). In a framework that supports first-class futures, this mechanism is automatic and transparent, but it could also be implemented specifically for the script interpreter. Finally, as the value of data is not needed in the script interpreter, communicating the value to the interpreter could be avoided (this would correspond to garbage collection of future references).

The mechanism consisting of transmitting a future reference and automatically updating the value has been formalized for ProActive and the ASP calculus in [34]. In practice, the ProActive framework partially provides what is needed in this particular case: futures are automatically created and transmitted (by copy) between objects or components, while future update is automatic. If, by relying on ProActive, the script interpreter were not to be blocked while waiting for the result of the first invocation, it could proceed to subsequent phases. However, the future update strategy currently implemented in the ProActive framework implies that all copies of the future are updated with the result; consequently, a copy of the data would also be sent to the script interpreter itself, even if it did not need this data.

4.2.5 Prospects for decentralized script evaluation

Another drawback of the script interpreter is that it provides a single central point for managing all reconfigurations of connections and data communications, which may become a bottleneck. This issue could be addressed in a hierarchical component model, such as the GCM, by encapsulating a script interpreter inside each composite component. An additional construct (*subscript*) would be added to the script language, delegating part of a script to the script interpreter associated with another component, and would consequently distribute the induced data communications more evenly.

Scalability would thus be ensured, but synchronization between those sub-script interpreters in a real distributed environment may be complex and is out of the scope of this study.

4.2.6 Alternative notation for composition in space

Currently, the proposed and implemented API for composition in space requires explicit invocation of *connect* methods on components. Although the notation proposed is as concise as possible, it would be feasible to use an alternative notation, similar to *imperative* programming, as in the case of composition in time. This would require adoption of decentralized script evaluation and imposing a specific convention on component code.

As a result of the following statement:

```
data = generator.getData()
```

the interpreter should not invoke the method, but only return a proxy to a *data* object with the dependency information. Subsequently, the following call:

```
result=simulator.simulate(data)
```

should be interpreted as a request to create a connection to the port through which the data can be retrieved:

```
simulator.inputPort.connect(generator.outputPort)
```

Finally, the *simulate()* call which would actually invoke *getData()* using the created binding.

4.3 Representation of components in GScript

As GScript is based on Ruby language, Ruby scripts are used to program the application, and, in addition to the standard Ruby library, special types of objects are introduced to represent activities performed on the components and other Grid technologies (see Fig 4.3). Since in general they may be mapped to any software entity (a component, service, legacy job, etc.) it was decided to refer to them as *Grid Objects*. The following vocabulary is used in relation to Grid Objects:

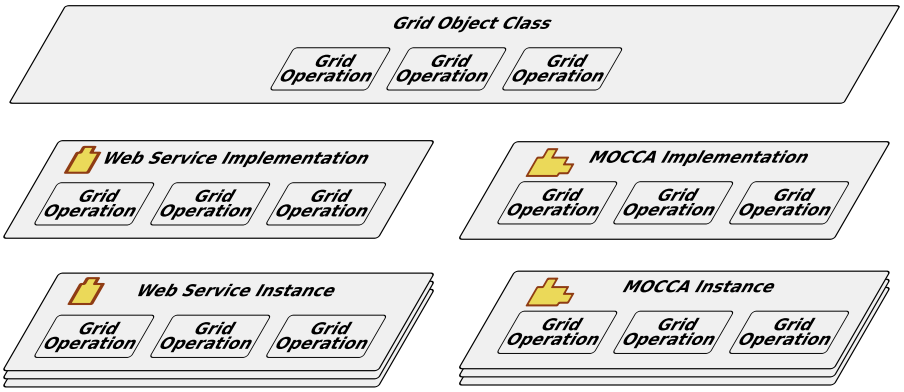


Figure 4.3: Hierarchy of Grid objects, implementations and instances

Grid Operation (GOp) , which corresponds to an abstract method in terms of object-oriented programming, which binds abstract methods offering given functionality. It is a method of a component port.

Grid Object Class (GOb) is an abstract class, or an interface, which declares a set of Grid Operations, and it is an abstraction of the same general functionality offered by different implementations. In terms of components, it corresponds to a component interface (port).

Grid Object Implementation implements functionality of its Grid Object Class; it is a static entity which has to be instantiated (deployed into a resource) to allow invocation of its operations. In the case of components, it is the component implementation.

Grid Object Instance is an instance of a certain Grid Object Class. Instances which use the same implementation may differ only in terms of resources they are deployed on. It corresponds to an instance of a component.

Grid Resource is able to host a Grid Object Instance. In the case of components, it corresponds to a component container.

Grid object abstractions allow interacting with many technologies providing access to computing in a transparent way. Of course, not all underlying technologies provide the same capabilities. Therefore some of Grid Objects are stateless, e.g. Web services, while others may be stateful (components). It is also possible to have a stateless Grid Object Class, with two implementations: one as a Web service, and another as a stateless component. The functionality of these implementations will be exactly the same, but the component may be dynamically instantiated on demand. This process is invisible to the script developer, but may provide an opportunity for optimization within the execution engine.

4.4 Architecture of the script execution engine

As the goal of the proposed Ruby-based GScript is to provide constructs for component deployment, composition and invocation of component methods, there is a need to provide an execution engine, responsible for providing information on components in use and hiding the complexity of the underlying Grid infrastructure. The architecture of the proposed system (called GridSpace) is shown in Fig. 4.4. The *Registry* is used for storing all technical information about available components, containers and the state of the Grid resources, updated by the *Monitoring* system. The registry is available as a Web service, and is also possible to use a local registry, which stores the same information in the Ruby script format, useful for quick development and debugging. The *Optimizer* module is responsible for supporting decisions on (automatic) component deployment. The role of optimizer is similar to the *deployment framework* as proposed in [39]. The *Invoker* module transparently performs remote calls on component ports of different frameworks.[1]

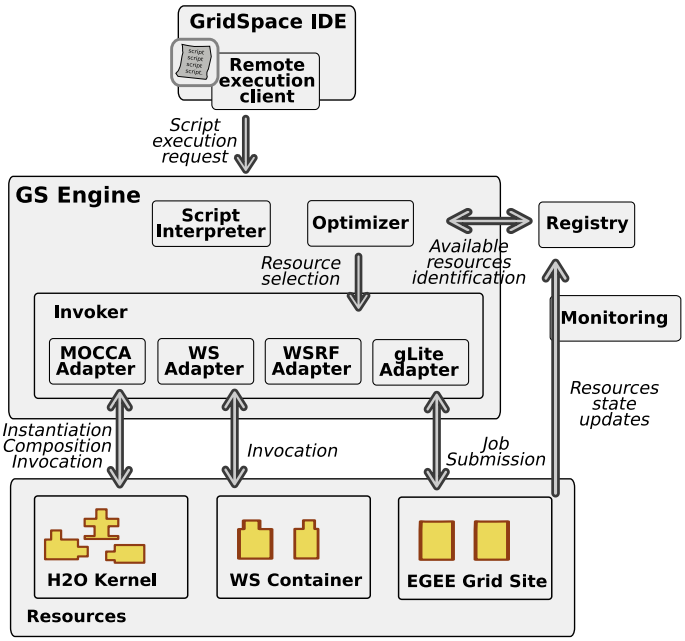


Figure 4.4: Architecture of the GridSpace scripting environment

The Invoker has an extensible architecture which allows plugging in adapters responsible for interacting with different technologies. As far as component technologies are concerned, there are adapters for communicating with MOCCA and

[1] The development of the Invoker module was performed by Tomasz Bartyński under supervision of the author [15, 115].

ProActive components. For Web Services, the adapter can use the client included in the standard library of the scripting language. There is also an adapter for the MOCCA-ProActive Glue components; those glue components allow composition in space (direct connections) between the two frameworks [118] (see Chapter 7 for details). More information on the Invoker and the architecture of the system can be found in [15].

Two types of information related to Grid Object Instances are required by GOI in order to realize its objectives. Firstly, it is the unique identifier of the optimal instance that should be used. This identification is used to retrieve further information, which consists of technology data that describes a concrete instance in terms of its communication protocol, the endpoint address, the interface and other technical details. These dependencies can be satisfied by either a simple local registry and optimizer or by remote systems.

Although programming in a scripting language, such as Ruby, is convenient, there is always a need to support the development process with user-friendly tools which assist in implementation and help reduce the number of mistakes. For GScript, an integrated development environment based on the Eclipse platform with Ruby Development Tools is offered[59] (see Fig. 4.5), enriched by additional plugins. One of them is the registry browser which lists all available Grid Object Classes, Implementations and Operations. It is connected with a script editor and allows one to automatically insert pregenerated code snippets, such as component creation methods. Another plugin may be used to browse the component classes categorized using an ontology-based taxonomy. It can be especially useful when searching for a component based on its functionality, and finding similar components.

4.5 Optimization in GScript

The source code of the script provides only information on Grid Object Classes – selection of actual instance is left to the engine. The choice of an instance, taking into account the structure of relations between the entities (Fig 4.3), is not trivial and entails the following decisions: *which* Grid Object Implementation will be the most suitable to perform the required processing; *which* existing Grid Object Instance of this Grid Object Implementation will be the most suitable; *whether* the Grid Object Instance should be chosen or whether a new one should be deployed; *where* (on which Grid Resource) a new Grid Object Instance should be created.

In traditional Grid environments, dedicated components – a *scheduler* and a *resource broker* – are provided to answer such questions. Since GridSpace calls for much broader functionality, the terms *optimization* and *optimizer* are introduced.[2]

[2]The development of the Optimizer module was performed by Joanna Kocot and Iwona Ryszka, under supervision of the author [122].

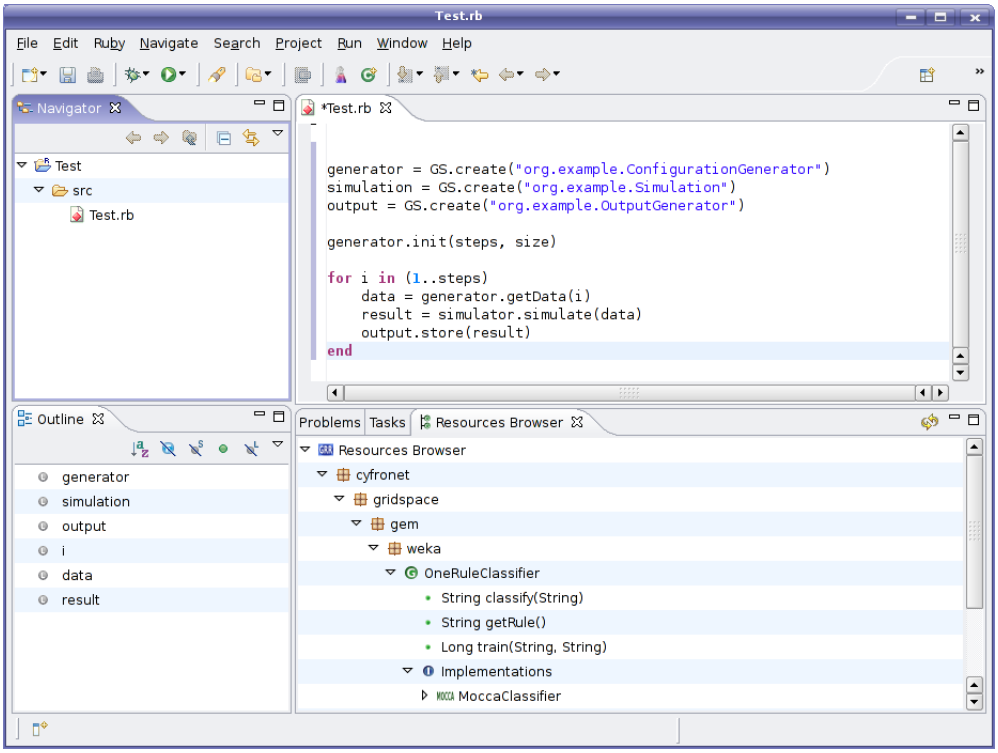


Figure 4.5: Eclipse Ruby Development Tools with sample script and Registry browser

Optimization of application execution in GridSpace can be presented as a special case of the generic optimization process in Grid computing [158]. Notions related to the GridSpace environment can be mapped onto the general Grid terminology in the following way: a Grid Object Instance represents a *resource*, a Grid Operation represents a *job*, a GridSpace script represents a Grid *application*. It is worth noting that there is no direct control over resources and the optimizer can only act as a broker; it cannot guarantee that the task it schedules will be executed on the selected resource and is not able to manage the tasks after they are submitted for execution. Access to resources is not exclusive and the information about resources gathered by the optimizer can be imprecise or out of date. Furthermore, the optimizer cannot guarantee that the performance of scheduled jobs will not be reduced by some tasks executed by the local scheduler.

The optimizer may be used in one of three modes with different dependencies between subsequent tasks taken into consideration:

- Short-sighted optimization, which implements the basic optimizer functionality: choosing an optimal solution for one task at a time.

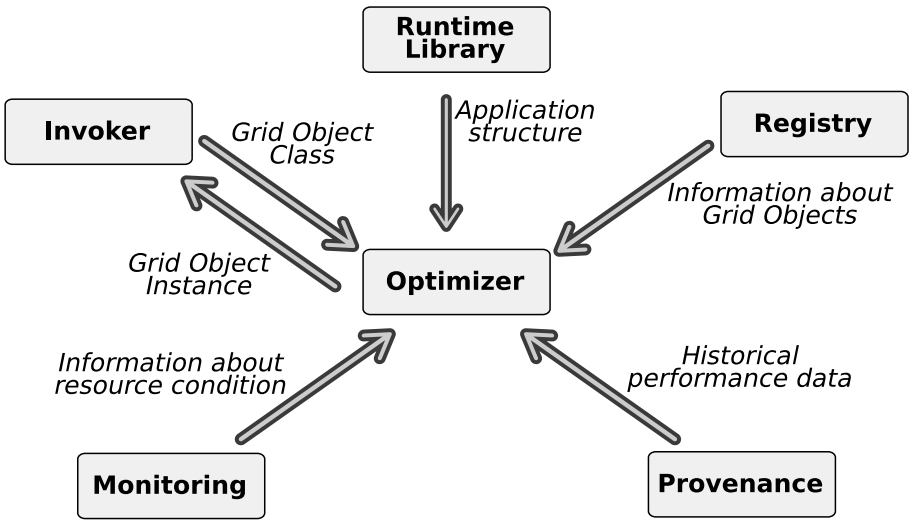


Figure 4.6: Optimizer placed in the context of neighboring components of the GridSpace engine

- Medium-sighted optimization, where tasks can be submitted to the optimizer in groups and for each task a resource is chosen on the basis of previous choices. The tasks are not reordered, nor arranged in queues and a group of tasks is mapped onto a group of resources.
- Far-sighted optimization mode, which is a suboptimal solution where resources are chosen for all tasks of an application and tasks may be reordered.

Thanks to the component architecture of GridSpace, the optimizer will be able to interact with other components which are part of the runtime and have access to the information gathered by components of the Middleware layer (see Fig. 4.6).

The only consumer of optimization results is the Grid Operation Invoker (GOI) [15]. It is responsible for creating Grid Object Instances or contacting existing ones, and invoking Grid Operations on them. The Invoker provides an identification of the Grid Object Class it is going to use, and obtains a Grid Object Instance or all the data necessary to create it.

The process of optimization, in an analogy to the schema introduced in [158] , begins with the resource discovery phase. A list of all appropriate Grid Object Implementations, including their running instances which are available for use at any given moment, is generated for a given Grid Object Class. The only mandatory source of information is the Grid Resource Registry (GRR) which contains generic and static data related to Grid Objects.

The system selection phase results in finding the best combination of a resource and a given job, i.e. determining the best Grid Object Instance on which the given

Grid Object Operation should be performed. The phase involves gathering dynamic information about resources from the Monitoring System. One additional, optional component is the Provenance Tracking System – PROToS [13], which can be a source of data related to the performance of Grid Object Instances in the course of their previous invocations. Additionally, for far-sighted optimization, data about the structure of the executed application (an *application graph*) is required. The source of such information is the core Runtime Library.

4.6 Conclusions

This chapter introduced the methodology of high-level programming of component-based applications on the Grid. This concept is based on a scripting approach and is intended to support rapid application development which may be useful for flexible construction of scientific experiments. The proposed scripting notation can be used to combine temporal and spatial composition of components, taking advantage of a modern scripting language with a rich set of control structures and object-oriented abstractions.

The novelty of the proposed approach, in comparison to scripting support existing in e.g. the XCAT framework, is that it provides a high-level interface which places focus on the application structure instead of low-level infrastructure details. Additionally, various underlying technologies (not just components but also services and jobs) can be seamlessly integrated into the application.

It should also be pointed out that the high-level composition and execution mechanisms can be compared to high-level workflow notations and their corresponding workflow engines, such as the one derived from the K-WfGrid [82]. The main difference comes from the usage of a scripting notation instead of a graph-based one (Petri nets). Experience with the applications (see. Section 9.4) suggests that a scripting approach is a valuable alternative solution, especially in the case of potentially complex applications which have to be developed in a rapid and dynamic experimental lifecycle.

The proposed notation is supported by a proper execution engine, responsible for translating high-level invocations into low level interactions with specific Grid middleware technologies. In this chapter, the architecture of such an engine is described, including Registry, Invoker and Optimizer modules. The Optimizer module is particularly important, as it may be crucial for efficient execution of applications on multiple resources.

A prototype of the above system was implemented and verified on a number of applications. Specific case studies and tests are described in Chapter 9. The proposed solutions were integrated and form a core part of the ViroLab virtual laboratory system [185].

The author's concepts of applying a high-level scripting approach to component composition were published in [120]. The concepts of GridSpace modules were also published: the Grid Operation Invoker in [15, 115] and optimization issues in [122].

Chapter 5

Application Composition Based on ADL

This chapter discusses an approach to composition and management of component-based applications which is based on the concept of a software Architecture Description Language (ADL). This high-level approach is an alternative to the script programming presented in Chapter 4, as it requires a precise descriptor of application structure, specifying spatial composition of components. While it does not provide the full flexibility of a scripting approach, its advantage is the possibility of using an automatic tool which can control the application lifecycle from deployment planning to execution and runtime management, in response to the changing conditions in the Grid environment. This chapter describes how this approach is applied in MOCCAccino, which introduces a flexible ADL and a corresponding manager system.

5.1 Introduction

Running component-based applications on the Grid remains a challenging problem, involving resource discovery, component deployment and application management in response to environment changes. While the scripting approach described in Chapter 4 can be successfully employed in such scenarios as rapid application prototyping and exploratory experiment programming, there are cases when the application is well structured and established, thus making it possible to specify its structure using a descriptor written in a software Architecture Description Language. A typical example would be a simulation, such as the one introduced in Fig. 4.1. The challenge is to efficiently map such an application to the underlying Grid infrastructure and to manage and adapt it to the changes of the environment at runtime.

The main characteristics of the problem may be briefly summarized as follows:

- The application components and their connections may have varying demands for computing power and communication cost;
- The application may include collections of components;

- The number of components in a collection may depend on the number of available computing nodes;
- The number of nodes may change over time.

The main assumption is that the component application should be unaware of the complexity of the environment it is running on. Thus, the responsibility for dealing with the deployment and adaptation process should be delegated to a specialized manager tool. Consequently, such a manager tool should be aware of both the environment and structure of the application. Environment awareness can be achieved by using appropriate discovery and monitoring techniques. The component application structure can be expressed using an Application Description Language (ADL).

5.1.1 The approach - ADL concept

The use of Architecture Description Languages is a known approach in component application programming, as discussed in Section 2.2.2. Despite the number of existing ADLs, none fully satisfy our requirements. There is a need to describe component characteristics (in terms of computational and communication requirements) or build structured component-based applications. Furthermore, specifying the number of components in a way that would enable adapting the number of components to the available resources is not supported.

Additionally, it is possible to rely on experience with the Automatic Flow Composer (AFC) tool [26], which facilitated application composition. AFC uses an application flow descriptor based on XML, which is a type of ADL specific to CCA components in the XCAT framework. The ADL used in AFC is also constrained in that it does not provide mechanisms to describe parametrized and hierarchical collections of components. For that reason, it was decided to design a new ADL notation.

Considering the advantages, disadvantages and limitations of the above mentioned ADLs, the following goals were identified when designing the ADL for MOCCAccino (ADLM):

- make ADLM as concise as possible,
- facilitate modification of quantitative architecture properties,
- not losing expressiveness of the language – supporting a wide breadth of application architectures,
- linking resources, such as executables and documents, using URLs,

- enabling containment of information related to other aspects of deployment such as planning or policies of adaptation.

To facilitate document processing and tool programming, an XML-based syntax was developed. The main elements of the description are presented below. *Component class* is a basic element that defines component types. It defines the specification of components, their behavior, ports and class names, to enable further creation and use of an arbitrary number of instances of each component class. The specification of component classes includes: component class name, fully qualified component Java class name, component Java archive file URL, *provides ports* description as well as *uses ports* description.

To facilitate the parametrization of application description, the term *component instances group* is introduced. *Component instances group* is established from an arbitrary number of component instances. These instances are physically separate and independent, but their connections follow an identical pattern. For example, in the case of a master-worker architecture, every single worker is connected with a master. Therefore, despite the fact that workers are individual, they are connected according to the same pattern. If so, they may be logically treated as a *component instances group* and represented as an ensemble in the *parametrized component diagram.*

When a component is connected to a collection, there is a need for an addressing scheme for indexing multiple connections. *Connection qualifier* defines the addressing scheme in a pluggable mechanism, and such basic qualifiers as *list* or *map* are provided. Connections are parametrized by qualifier-specific properties such as list length or map key set.

An important rule in the system is that connection multiplicity implies how many component instances of a group are to be created. For example, if a master component is connected to the workers collection using a list qualifier, then the number of workers will be equal to the list length. In other words, connection multiplicity defines the multiplicity of component instances and it is the role of the MOCCAccino system to ensure that a proper number of components will be instantiated and correctly connected.

Deducing the required number of component instances also depends on the connection type. When the connection is of the *separate* type, then for each user-side component a number of provider-side components are instantiated and connected. Hence, each provider-side component is connected to only one user-side component. Alternatively, in the case of a *shared* connection, a number of provider-side components is instantiated in such a way that every provider-side component is connected to each and every user-side component.

Obviously, since the *parametrized component diagram* does not have to be a simple tree, connection cycles may exist. This may result in ambiguity of the number

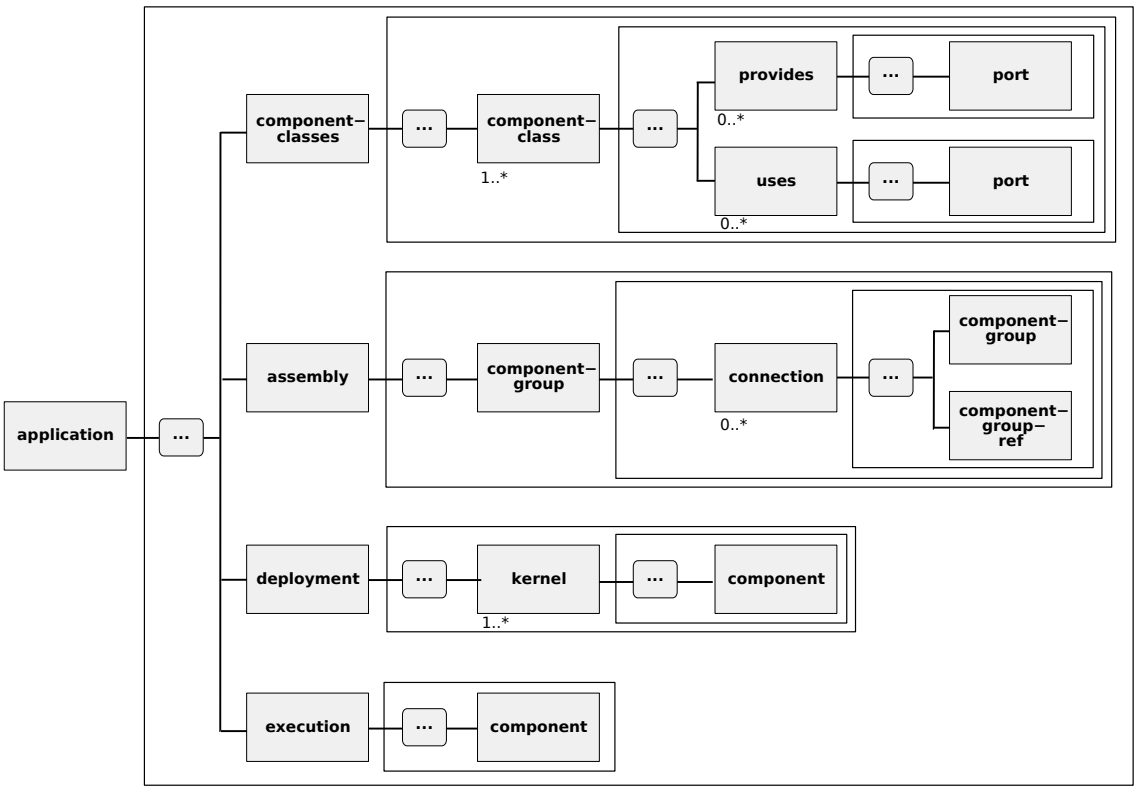


Figure 5.1: Visualization of the ADLM XML Schema

of component instances. This issue (among others) is covered by the Component Graph Builder (CGB) which validates the diagram.

The MOCCAccino Component Graph Builder (CGB) validates the *qualitative component diagram*, resolves it, and returns a corresponding plain component graph, which is assembled from plain component instances.

ADLM expresses a *parametrized component diagram*. Such a diagram comes with an object representation called the Application Object Model (AOM) and may be built by the AOM API provided.

The textual representation of AOM is ADLM, which is XML-based and is handled by the MOCCAccino ADL Marshaller. The ADLM schema is presented in Fig. 5.1.

MOCCAccino can also support ordinary (unparametrized) component diagrams by describing all connections as individual (with multiplicity equal to one) and then treating the *component instances group* as a single component instance.

It is also worth noting that a number of MOCCAccino-specific decorations are present in ADLM. These include attributes associated with the *component instances group* and connections which provide information about computational power re-

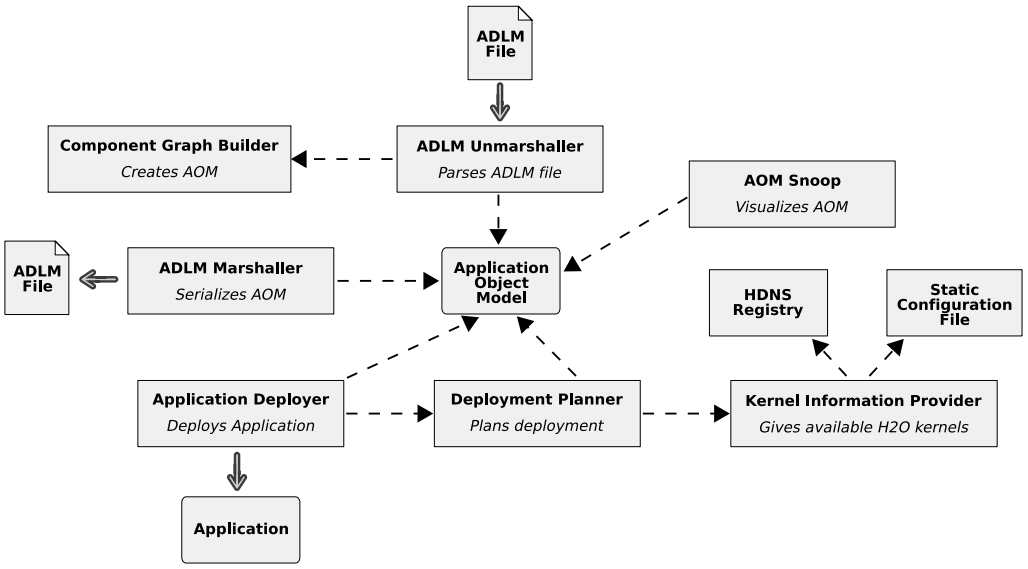


Figure 5.2: MOCCAccino components with their dependencies

quirements and connection load which are used during deployment.

5.2 MOCCAccino manager system

5.2.1 Architecture of MOCCAccino

In order to validate the proposed ADL approach, the MOCCAccino Manager was designed.[1] Fig. 5.2 presents the dependencies between MOCCAccino components along with results of their activities.

The MOCCAccino Manager is responsible for deployment, execution and management of the application. Its functional components are: ADLM Unmarshaller, Component Graph Builder (CGB), Kernel Information Provider (KIP), Deployment Planner (DP) and Application Deployer (AD).

The ADLM Unmarshaller component is responsible for parsing the file that contains the application description ADLM file (XML-based). While parsing, the application model is built using the Component Graph Builder to create a component graph. The model that represents the application is called the Application Object Model (AOM). It is a data structure which can be exchanged between functional components to construct and deploy the application.

[1] The implementation of the MOCCAccino system was performed under the author's supervision by a team of students (Eryk Ciepiela, Joanna Kocot, Tomasz Bartyński and Przemysław Pelczar), co-authors of papers [117, 116]

The deployment process itself is handled by the Application Deployer component. The result of his action is a deployed application and its handle, which the Manager can use. The Application Deployer uses a Deployment Plan, which assigns a H2O kernel to each of the application's MOCCA components. This plan is built by the Deployment Planner, which can implement different policies to optimize the plan. The H2O kernel information required for construction of the Deployment Plan is obtained from the Kernel Information Provider component which may be configured to run in static mode (by providing a list of available kernels) or dynamic mode, in which case it uses the HDNS [72] registry.

In the latter case, the Kernel Information Provider subscribes in the HDNS to events concerning appearance and disappearance of resources in the environment. Subsequently, the notification is delegated to the Runtime Manager which deploys new components as necessary. The choice of component to be deployed bases on the policy which is either implemented in components (environment-aware components) or externally defined in the ADLM file (environment-unaware components).

MOCCAccino also provides a tool for visualizing the application model, expressed either in AOM or ADLM forms.

Fig. 5.3 presents the activities of the MOCCAccino Manager, along with external interfaces, results of their activities and the order in which activities are undertaken.

5.2.2 New CCA extensions

Since all MOCCA components are to be manageable, with their ports connected externally and dynamically by the Manager, each component has to be equipped with a special dedicated port.

The *Configuration Port*, provided by MOCCAccino, can be accessed remotely by the Manager via an interface named *ConfigurationPort* which is accessed locally by the owner component via a local interface called *ConfigurationPortLocal*.

Each MOCCAccino-compliant component has to register a *Configuration Port*, which will be used to register, connect and resolve uses ports. The component calls *getQualifierByName()* to receive the qualifier of a multiple connection. Sample usage of the *ConfigurationPort* is depicted in Fig. 5.4.

The *Configuration Port* is used not only at the assembly stage, but may also be considered a management port of the running component. For example, in the case of dynamic appearance of available H2O kernels, a corresponding event has to be intercepted in the *Configuration Port*, and then processed according to the component's logic.

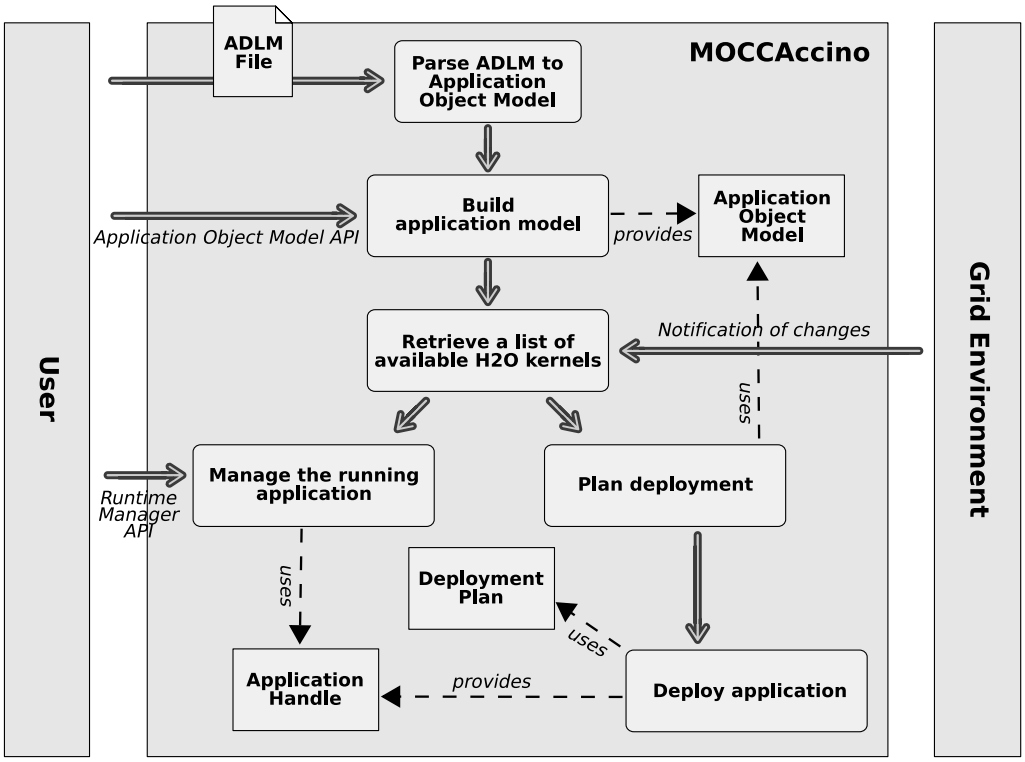


Figure 5.3: MOCCAccino Manager activities and control flow diagram

5.2.3 Deployment planning and application management

In order to enable application deployment alignment with the Grid environment the following characteristics and metrics of the infrastructure as well as of the application's architecture have to be introduced. Such metrics concern computational power provided by containers and, likewise, required by the component instance as well as network bandwidth available between containers and connection load between component instances.

Classification of resource requirements introduced by MOCCAccino includes:

1. Computation power required by component
2. Network bandwidth required by connection

To fulfill these requirements, MOCCAccino introduces such parameters as *component weight* and *connection weight*.

Weights are to be tuned arbitrarily by the application deployer, and their values are examined relatively to each other. *Component weight* specifies the computational power required by a component, while *connection weight* expresses network

```
public void setServices(Services services) throws CCAException {
  // Moccaccino components must be equipped with configuration port...
  this.configurationPort = new ConfigurationPortImpl(services);
  // ...
}

public void foo() {
  // ...thus, Moccaccino component can resolve dynamically and access ports
   ListQualifier pongs = (ListQualifier) this.configurationPort
     .getQualifierByName("pongs");

   for (int i = 0; i < pongs.getLength(); i++) {
      PongPort pongPort = (PongPort) configurationPort
        .getUsesPort(pongs.getPortNameAt(i));
      pongPort.foo();
   }
  // ...
 }
```

Figure 5.4: Sample usage of *ConfigurationPort*

bandwidth required by a connection. Subsequent deployments may be tuned by manual manipulation of weight values.

Deployment planning is based on a model which follows two fundamental principles: computation weight repels components, thus strongly repelled components are deployed on different H2O kernels. Conversely, weighty communication attracts components and as a consequence, strongly attracted components are placed in the same kernel. Through appropriate adjustment of *connection weight* parameters, it is possible to achieve collocation of tightly-coupled components or distribution of loosely-coupled ones. Simultaneously, the deployer may avoid co-allocation of computationally-intensive components through assignment of an adequately high value to the *component weight* parameter, whereas fine-grained low-weight components are still allowed to co-exist in the same container.

In MOCCAccino, weights may be specified in the ADLM file as well as via the AOM API. In both cases, weights are attached to the *component instances groups* and to the *connection group*.

5.2.4 Optimization of deployment planning

Optimization of deployment planning in the model where component weights and connection weights are specified, can be expressed mathematically as a problem of finding an optimal mapping of the components to the available nodes.

To demonstrate that such a model can lead to reasonable planning, the author shows how it behaves assuming the simplest homogeneous infrastructure. For simplicity's sake, it can be assumed that the nodes are homogeneous in terms of computing power. Network connections between nodes can also be treated as equal.

However, it is possible to allocate more than one component on a single node (co-allocation). In such a case, the connection between these components is local and can be assumed to be considerably faster.

Let us introduce the following notation:

- $Conn(i)$ – the weight of the connection i,
- $Comp(i)$ – the weight of the component i,
- $Node(i)$ – the number of the node component i is mapped to,
- $Plan(i, j)$ – the planning matrix representing the weight of component i on node j.
- $L(i) = \begin{cases} 0, & \text{if connection } i \text{ is local} \\ 1, & \text{if connection } i \text{ is remote} \end{cases}$

Each component can only be allocated to a single node, but there may be more than one component on a node. Hence, the planning matrix has the following property:

$$Plan(i, j) = \begin{cases} Comp(i) : Node(i) = j \\ 0 : Node(i) \neq j \end{cases}$$

Consequently, the function representing the total cost of the mapping can be defined as the following sum:

$$Cost(Plan) = \sum_i L(i)Conn(i) + \max_j(\sum_i Plan(i, j))$$

Below it will be shown that such a relatively simple model of weights can result in optimal mappings in most common scenarios such as task farm or domain decomposition, yielding such results as load balancing and co-allocation of tightly-coupled components.

Task farm

In this scenario, it is assumed that there is one *Master* component and a group of *Slave* components, and all the slaves are the users of a single port provided by the master component (see Fig. 5.5). The weight of the master component is m and the weight of the slave component is s. It is assumed that all connections have the same weight, equal to 1 (for simplicity's sake).

Assuming that the number of nodes is N, the optimal mapping is to allocate one component per node. In this case, the total cost of a plan is equal to:

$$N + \max(m, s)$$

It can be seen that this plan is optimal, since allocating e.g. two components to one node will double the weight of that node, increasing the total cost.

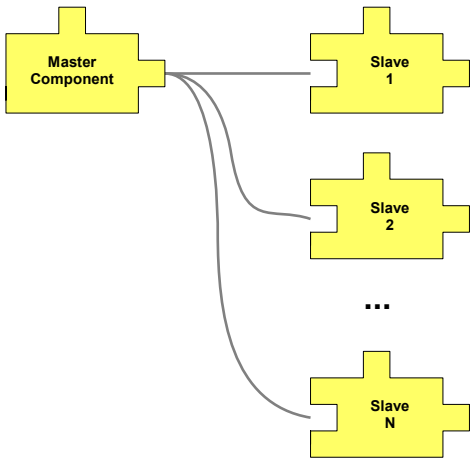


Figure 5.5: Task farm scenario modeled as component assembly

2D domain decomposition

In this scenario, typical for domain decomposition applications, there is a number of components organized in a Cartesian grid, each communicating with its neighbours. In order to decouple the computational part from the communication part of the application, two types of components are introduced: *CommunicationComponent* and *ComputingComponent*. The communication components are reusable for any type of domain decomposition applications and are responsible for synchronization and communication between the computing components. The computing components perform the actual calculations in the steps coordinated by communication components. For further details of the domain decomposition example, please refer to Sec. 9.5. In the case of two-dimensional decomposition, a sample assembly is shown in Fig. 5.6.

The natural mapping of components to homogeneous nodes should co-allocate computational components together with communication components and, at the same time, distribute computing between nodes. Denoting the weights as follows:

- C – computing component weight,
- L – the weight of links between communication components,
- I – internal communication between computing component and communication component,
- 1 – the weight of communication components,

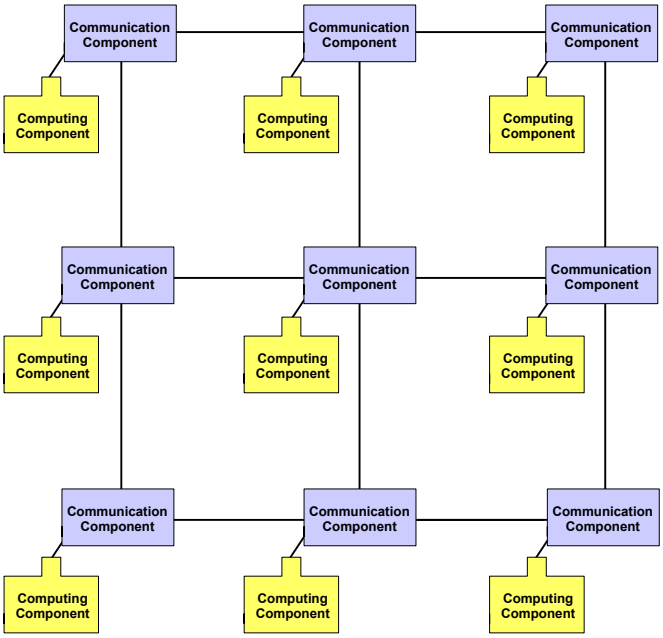


Figure 5.6: Domain decomposition scenario with decoupled communication and computation components.

yields the formula for the total cost of a plan on N^2 nodes (usually, it is assumed that $I >> L$):

$$Cost(Plan) = (C + 1) + 2N(N - 1)L$$

It can be seen that such a mapping is optimal, since if it is changed, the cost increases. If any of the communication components are moved from the corresponding computing components, weight I is added to the sum, increasing the total cost. If M computing and communication component pairs are co-allocated on the same node, the first term in the equation increases to $M(C + 1)$, also increasing the total cost.

Optimization – discussion

As can be seen, the proposed optimization planning model is simple and the goal of presenting it here is merely to demonstrate that the proposed ADL notation and the manager model enable construction of an overlaying optimization layer. It was shown that even such a simple model (with a homogeneous infrastructure) can reproduce such properties as load balancing and co-allocation of components.

Obviously, more advanced models may take into account the heterogeneity of the infrastructure and more complex cost functions. It would then be possible to use appropriate optimization methods and heuristics to calculate an optimal planning matrix. For this purpose, co-allocation scheduling techniques can be applied [30, 95]. This requires more specific input from the infrastructure, together with the requirements of the components, and points to interesting research directions, which, however, are out of the scope of this thesis and are left for future work.

5.2.5 Handling dynamic changes of the environment

The proposed MOCCAccino manager system is also designed to handle dynamic changes in the environment. Such changes may occur if the pool of nodes (H2O kernels) is created dynamically on some existing Grid infrastructure (see Chapter 8 for details). In this case, the system should behave according to the autonomic computing loop of control [103]. First, monitoring should detect the change in the environment (new kernel is added to the pool). Subsequently, a reconfiguration decision has to be taken, according to some policy. Finally, the decision should be executed by the manager system.

Regarding monitoring, in the case of MOCCAccino, resource information is stored in the HDNS registry and is provided to the manager by the KIP module. As HDNS does not provide notification on resource changes, a monitoring agent was introduced into the KIP, which periodically checks the HDNS registry to detect new information. If a new kernel is detected, the KIP notifies the manager about such an event.

To define a policy which governs the behavior of the application in case of a change in the environment, the ADLM notation was extended. An additional component group *qualifier* was introduced, which indicates that there should be one component *per kernel* in such a group. This will be useful especially for compute-intensive components forming a task farm. In this case, the worker components should occupy all available kernels and new ones should be added to the application once a new kernel joins the pool. To resolve any potential conflicts, there is the constraint that only one group may exist with such a *per-kernel* qualifier.

For executing reconfiguration, a *dynamic application handler* class was introduced. It receives the change event from the KIP and, using the information from AOM, creates a new component in the new kernel. The component is added to the group and connected to existing components, as described in ADLM. Similarly, the component is removed when the monitoring detects the removal of a kernel.

The presented reconfiguration scenario demonstrates the usefulness of having an application described using a parametrized ADL and represented at runtime by AOM. The runtime manager API allows management of the application and plugging in external, possibly autonomic controllers.

5.3 Conclusions

The goal of this chapter was to present an approach to application composition based on the Architecture Description Language concept. A new ADL was defined, which is specifically suited for CCA components and its goal is to facilitate deployment and management of computationally-intensive applications on the Grid. The proposed ADL allows specifying components organized in hierarchical groups, with parametrized numbers of members and flexible connections described using qualifiers. Consequently, the author proposed the architecture of the the MOCCAccino manager system, which can process the XML-based ADL description and deploy the application on the available resources. Mechanisms for adaptation to changes in the environment are also introduced in the system.

To show that the proposed approach enables automatic deployment plan optimization, a sample model was introduced, which is based on the weights of components and connections. It was demonstrated how this model can be used to reproduce such common scenarios as task farm and domain decomposition with load balancing and co-allocation of components. It was also shown that by having an application described using a formal ADL notation, it is possible to introduce adaptive behavior which supports automatic management of an application operating in a dynamically changing environment.

It must be emphasized that the methodology proposed in this chapter should be considered complementary to the scripting approach described in Chapter 4. The ADL-based approach is better suited for applications which have an established structure and which are built using the composition in space method. In this case, the user can obtain better planning optimization and autonomic management features. On the other hand, ADL does not provide the flexibility offered by the scripting notation and composition in time is not possible. The author is convinced that providing both types of high-level composition – ADL and scripting – is equally important for the component-based programming and execution environment if it aims to support a wide range of scientific applications.

All the MOCCAccino modules were unit-tested and integration testing was also performed. The system was validated on a number of sample ADL files representing various application structures. The concepts described in this chapter were presented in [117, 116].

Chapter 6

MOCCA as Base Component Environment

In Chapters 4 and 5 such terms as CCA component, H2O kernel, component creation and connection were often used. In the proposed environment these fundamental capabilities are provided by the underlying component framework, called MOCCA. This chapter describes the basic concepts and assumptions of MOCCA and shows how they help fulfill the requirements imposed on the environment by scientific applications. The goal is to demonstrate that combining the features of CCA model and the advantages of the H2O platform can result in a powerful and flexible component framework, which can serve as a basis for the high-level programming capabilities discussed in the preceding chapters.

6.1 Introduction

According to the requirements of scientific applications and the nature of Grid infrastructures, in Chapter 1 the desired features of the programming and execution environment were identified. Following analysis of the state of the art, the author's concept presented in Chapter 3 is based on the need of a distributed component framework, combining the advantages of the CCA component model and the H2O model of resource sharing.

In this chapter the author provides a detailed discussion of the concepts of the MOCCA component framework. The most important capabilities which the framework must provide are the *deployment* and *execution* of component code on the shared resources, a *communication* mechanism, which should be adaptable to the Grid environment and *flexibility* of application creation and adaptation.

The purpose of the following sections is to discuss in detail the goals of the framework, then to present the main concepts and premises underlying the design, and finally to discuss the proposed extensions to the CCA model which can facilitate the creation of higher layers of the environment.

6.2 Goals

The aim of the author's work is to design and implement a CCA-based framework for the H2O platform. The framework should fulfill the following goals:

- Provide easy mechanisms for creation of components on distributed shared resources;
- Provide efficient communication mechanisms;
- Allow flexible configuration of components and various application scenarios;
- Support native components, i.e. components written in non-Java programming languages and compiled for specific architectures.

As outlined before (see Section 2.3.3), the H2O platform possesses several features that can provide strong benefits for the distributed CCA framework. They can be enumerated as follows:

- H2O enables dynamic installation of software components on shared resources by deploying pluglets in H2O kernels. Such a mechanism will allow easy and dynamic creation of CCA component instances on distributed sites.
- H2O uses RMIX as an efficient communication layer. Since the CCA ports communicate in an RPC style, RMIX is well suited to connecting remote ports. When the components run on the same machine, it is possible to exploit this locality instead of using the network for communication.
- H2O allows multiple usage scenarios and component configurations, which can be directly exploited in the CCA framework: it enables separating the resource provider from the deployer of the component, by facilitating deployment of component code on shared resources. Although many users may deploy their components in one H2O kernel, they do not interfere with one another owing to the H2O security mechanisms. Thus, users can create their own distributed arenas on overlapping sets of resources in a secure way and exploit this locality wherever possible. Moreover, H2O separates the roles of component deployer and user. For instance, a third party reseller may deploy a stateless component as a persistent service, allowing users to connect to its ports and communicate with it.
- If required, it will be possible to enable compatibility with Web services (as is the case with XCAT) by reusing mechanisms of the Web service pluglet [164].

All these features imply that the H2O platform, through its component foundations and orientation on metacomputing, has sufficient capabilities to provide a foundation for a powerful and efficient CCA framework, targeting metacomputing and Grid environments.

6.3 Concepts and design

This sections presents the design choices for the developed CCA framework, stemming both from requirements and from specific H2O features.

Firstly, it was decided to have one-to-one mapping between a CCA component and an H2O pluglet. This design choice makes the deployment of a component in the framework as easy as the deployment of a pluglet. Moreover, it takes advantage of the fact that H2O each pluglet operates in a secure execution environment. This is achieved by the H2O kernel, which uses separate Java class loaders for different pluglets, and adapts Java security mechanisms to enforce protection boundaries. Thus, placing each CCA component instance in a separate pluglet enables secure operation of many such instances on a shared resource. Even multiple, different versions of the same component can run at the same time. The challenge related to enclosing components in pluglets is that it makes exploiting locality more difficult. Method calls between components in the same kernel are, by default, executed as remote calls. However, RMIX allows applications to switch to a local binding (utilizing a shared address space), subject to access control policies. Since this approach compromises component isolation, the choice of strategy is left to the application composer.

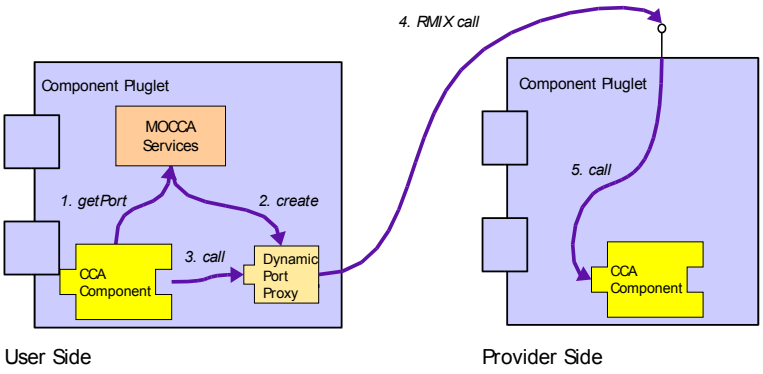


Figure 6.1: Usage of dynamic proxies for connecting CCA ports

The second issue is the remote communication mechanism. It would be convenient to design CCA Port interfaces as RMIX remote Java interfaces

(`java.rmi.Remote`). However, to be compliant with the CCA specification defined in SIDL, it is necessary to take into account the fact that the Java bindings to SIDL interfaces defined in Babel impose a specific inheritance hierarchy. Thus, it is not possible to straightforwardly turn them into RMI-compliant remote interfaces. The solution to this problem relies on introduction of an intermediate layer which translates invocations of CCA-based interfaces to RMI remote interfaces. In the client-side (*uses* port), this layer is represented by a *dynamic proxy*. The server-side (*provides* port) exposes a remote interface of the MOCCA Pluglet, which includes the `invoke()` method. This method delegates to a call of the CCA-compliant *provides* port implementation, thus making the intermediate layer fully transparent to both component developers and users. The interaction between the component and the framework is shown in Fig. 6.1. Assuming the *user* component is connected to the *provider* component, the *user* needs to invoke the `getPort()` on the framework services. Following this, a port proxy is created and returned to the component. Subsequent proxy calls are delegated to the remote *provider component* (see Fig. 6.2).

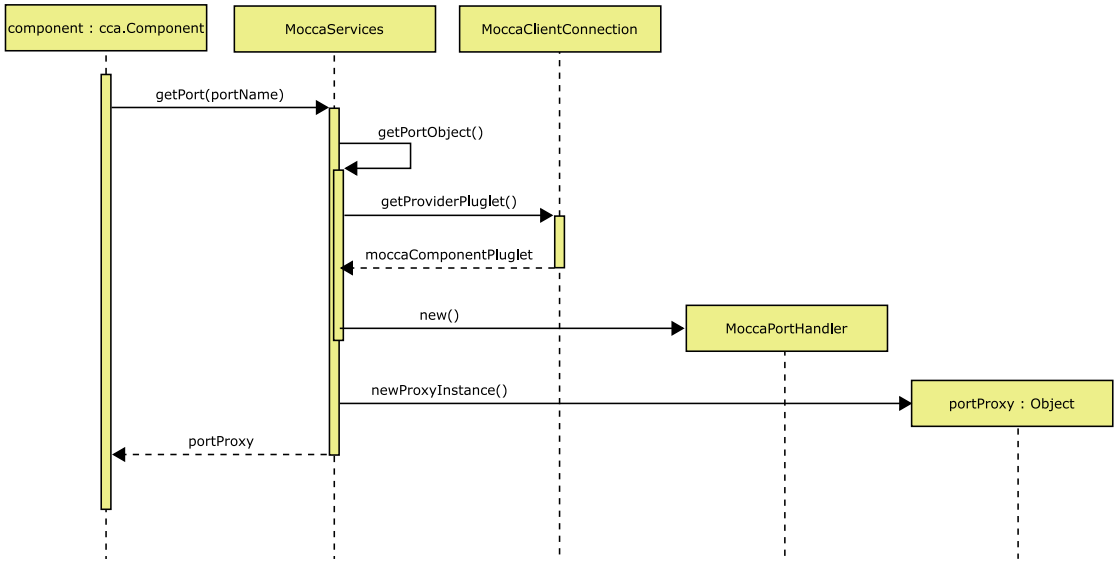


Figure 6.2: Detailed sequence diagram for obtaining a reference to a remote uses port. When calling *getPort()* on a framework *Services* implementation, a remote reference to the *provider* component is obtained from the *MoccaClientConnection* object, which is passed to a *port handler* responsible for handling method invocations on a newly created *dynamic proxy* object.

In the case of a distributed CCA framework, it is necessary to decide how the CCA *BuilderService* port is to be designed. The Builder is a main interface that the framework exposes to the user and it allows creating component instances, con-

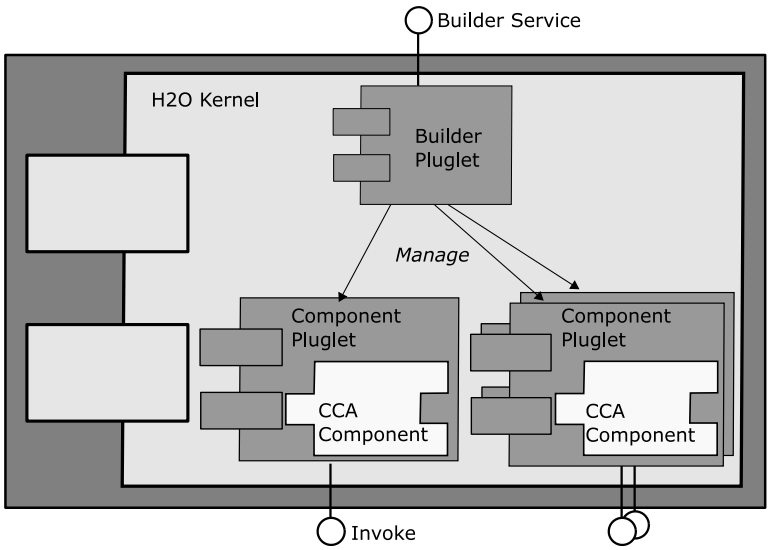


Figure 6.3: Deploying CCA components in the H2O kernel

necting them and querying their state. In MOCCA, it was decided to create a hierarchical builder system. MoccaBuilder is a pluglet which can create new components and manage them within the kernel in which it is deployed (see Fig. 6.3). In order to create components in many different kernels, it is necessary to first deploy Builders in each kernel. To minimize the distributed state, information about connections is kept in components that use ports provided by other components. Clients can build their own distributed component arenas by using several Builder pluglets deployed in different H2O kernels. Owing to the security mechanisms of the H2O kernel, many different users can deploy their own Builders to the same kernel at the same time and create their own component arenas, without interfering with one another, as shown in Fig. 6.4. Overlapping arenas can also be created, e.g. when collaboration between different groups working on the same project is required.

To facilitate usage of the system by individual users, who need to quickly compose applications consisting of some components and instantiate them on kernels which they have access to, an additional builder service is introduced. *MainBuilder* constitutes an entry point for the client, providing a simple API to deploy new Builder pluglets and the standard builder API to compose CCA applications. MainBuilder forwards creation and connection requests to the appropriate builders (see Fig. 6.4). It is possible to give the user full control over the placement of newly created components. Alternatively, MainBuilder may support pluggable load balancers which would automate component placement based on some optimization criteria.

One design issue which has been resolved in an almost automatic way, involves

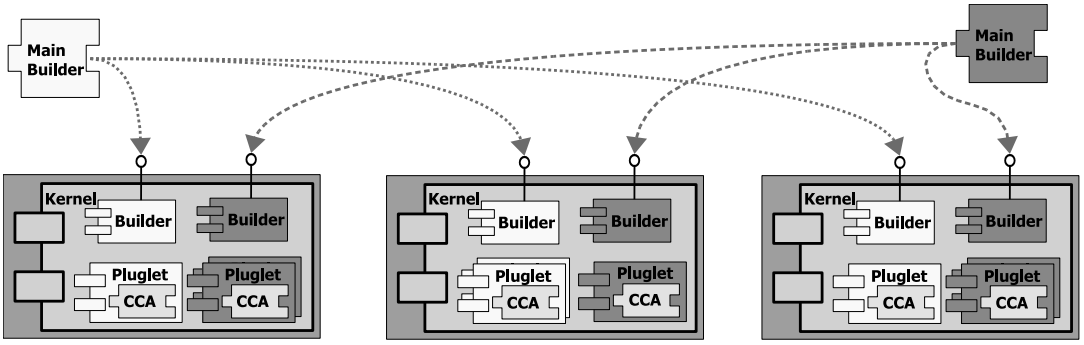


Figure 6.4: Multiple users can use the same resources.

the nature of ComponentID, which is used by the CCA framework as a unique identifier of the component. It was decided to use the H2O pluglet URI for this purpose. This URI can be passed from one kernel to another and, given this URI, the client (e.g. the component initiating the interaction) can invoke a remote method on the pluglet (e.g. a method of the provided port). By exploiting the hierarchical composition of H2O pluglet URIs, it is easy to decompose them into a kernel part, a builder part and a component part (e.g. `http://my.kernel/mybuilder/mycomponent` facilitating management of framework components.

In order to use the framework, it is necessary to have clear and convenient user interfaces enabling the construction of applications by deploying components and connecting their ports. The first interface to the framework is the MainBuilder Java interface, which implements the CCA Builder service port. Framework-specific information, such as the location of H2O kernels where the component is to be deployed, is passed in the properties, making this interface fully CCA-compliant. In addition to the Java interface, MOCCA offers a Jython scripting interface, similar to the one found in XCAT. Jython [100] is a Java-based implementation of the Python language and it allows interaction with Java objects nearly in the same way as with regular Python objects. Consequently, the MainBuilder API is also accessible from a Python script. It is possible to use the interpreter to interactively communicate with the framework, which makes development and prototyping of component applications very convenient. In addition to Jython, MOCCA distribution includes a JRuby [96] interperter. No GUI tool has been developed yet, but this type of interface is left for the upper layers.

6.3.1 MOCCA – conclusions

The goal of MOCCA was to create a CCA framework operating on top of the H2O platform and this goal was achieved, resulting in a flexible, easy to use and relatively

lightweight implementation of the CCA specification[1]. Conclusions from the work on developing MOCCA are that H2O is a very convenient platform for creating such a CCA framework targeting metacomputing or Grid environments. A pure-Java MOCCA implementation (sometimes referred to as MOCCA_Light) has proven a flexible and useful framework which can serve as a basis for other, higher layers of the component-based environment, as proposed in Section 3.3 and outlined in Fig. 3.1.

One of the goals of MOCCA was to be as lightweight as possible in terms of maintaining the state of the system. This was made possible thanks to the proposed hierarchical structure of builders and no central point in the architecture: each application of each user can construct its own instance of the running framework. Such instances can span different but possibly overlapping resources, they do not require any centralized control, and, moreover, they do not have to be aware of the existence of others. Such issues as coordinated management or maintaining a common view on the system are left for the upper layers.

Not all goals of MOCCA could be achieved in its Java-based implementation. Support for components written in multiple programming languages with the usage of Babel tool is described later, in Chapter 7. Another set of extensions, which tries to overcome the limitations of the current CCA specification, includes parallel component constructs, described in the following section.

6.4 Approach to parallel constructs

One of the features of the component-based environment proposed in Section 3.3 and outlined in Fig. 3.1 is to support parallel programming structures. In Grid computing scenarios for scientific applications it is often necessary to simultaneously run multiple components of the same type across many distributed resources. The goal of this section is to present an approach to introducing support for parallel constructs into the CCA model, which can be used without much effort in typical applications.

The proposed strategy is based on exploiting the dynamic runtime nature of CCA components [63], which enables deferred definition of component interfaces. In CCA, a component may add a *uses* or *provides* port during application execution, when required by runtime conditions. This feature is used to define components with a variable number of ports of a single type. Such *multiple uses* ports are suitable for constructing gather-scatter topologies of components. Another feature that can be leveraged is the connectivity between many users and a single *provides* port.

Fig. 6.5 shows an example of multiple components connected to a single *provides* port. The server component is not aware of the actual number of clients connected

[1]MOCCA implementation is available at `http://www.icsr.agh.edu.pl/mambo/mocca`

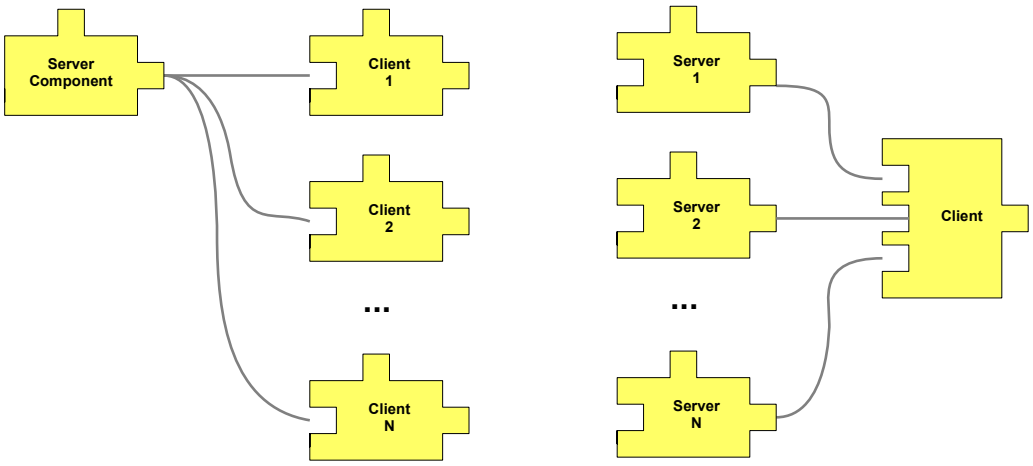


Figure 6.5: Example of multiple ports and components. It is possible to connect multiple clients to a single *provides* port (left figure), whereas it is necessary to create multiple *uses* ports in a component, which is a client of multiple providers (right figure).

to its port and handles their invocations in the very same way as if they were coming from one component. When there is a need to connect a client component to multiple providers, the client must have one *uses* port for each provider. As CCA allows components to register *uses* ports to the framework at runtime, their number doesn't have to be fixed – instead, it can be parametrized or even dynamic.

A mechanism to facilitate such parametrized multi-port components has been defined. Following component instantiation, the number of ports can be set as a property using *ParameterPort*. Subsequently, the component registers its multiple ports with the framework, assigning them names which (by convention) end with consecutive numbers. Such multiple ports may be then connected, in a simple way, to multiple components. To enable this functionality, the *BuilderService* interface was extended to handle such multiple components and connections. Examples of such *MultiBuilder* extensions are shown in Fig. 6.6. It is worth noticing that such an approach can lead to future possibilities for more dynamic changes of port numbers at runtime, e.g. in response to changes in the infrastructure within which the application is running.

The proposed approach does not provide a collective operation call, but a *user* component is responsible for handling single invocations on a collection of ports. This is more appropriate for scenarios when e.g. a master component individually controls its worker components independently of each other, rather than invoking a

```
public ComponentID[] createMultipleInstances(
          String instanceName,
          String className,
          TypeMap properties,
          ComponentID[] builders) throws CCAException

public ConnectionID[] connectMultipleProviders(
          ComponentID user,
          String usingPortName,
          ComponentID[] providers,
          String providingPortName) throws CCAException
```

Figure 6.6: Selected extensions introduced in the *MultiBuilder* interface

broadcast/scatter operation at regular intervals. Therefore, the proposed approach does not deal with automatic data redistribution, as in GridCCM or GCM, but the solution is more similar to a collective client port in Fractal/ProActive[11]. The `MxN` composition model [20] is not supported either, which poses an interesting research problem, solved e.g. by DCA [19] or GridCCM [148] frameworks, but of lesser practical value.

The CCA model often suggests the *single component multiple data* (SCMD) model in analogy to the known SPMD model of parallel computing [4]. In this approach, the aspect of parallel decomposition of components is orthogonal to the decomposition of application into components. This means that each of the application components may be parallelized, i.e. each may consist of several parallel processes communicating e.g. using MPI. The inter-process communication within a single parallel component is not specified by CCA and is left for a specific implementation of the framework or at the application level. MOCCA does not support any specific intra-component communication or parallelization mechanisms; there are, however, no limitations on a single component. Such a component can e.g. create multiple threads which exploit the multicore processor which runs the H2O kernel, or it may just be a wrapper for some MPI-based parallel program. On the other hand, the *multibuilder* extensions, as described in this section, aim to facilitate creating and managing the collections of components in practical master-worker scenarios, typical for Grid applications and used by the applications described in Chapter 9.

6.5 Summary

This chapter was devoted to a detailed description of the goals and design considerations of the MOCCA component framework, which forms a base layer of the proposed programming and execution environment for scientific applications. By combining the CCA model and the unique features of the H2O platform, it was possible to create a framework which enables dynamic deployment of components

on distributed shared resources, efficient communication and support for flexible scenarios. MOCCA was designed to be ready for integration with Web services and Babel-enabled CCA components. MOCCA also provides extensions for creation of parallel collections of components in a simple way.

All these features make MOCCA an innovative CCA implementation, making it very convenient for use not only directly by applications, but also as a basis for higher layers of the environment which is the subject of the research described in this dissertation. In light of the fact that interfaces need to be simple and compliant with the CCA specification, it has proven very convenient to use MOCCA as the base for the higher-level tools described in Chapters 4 and 5, as well as for integration and interoperability with other component-based solutions and infrastructures as shown in Chapters 7 and 8. Moreover, the tests described in Chapter 9 favourably compare MOCCA performance to alternative implementations of CCA and demonstrate its applicability to a wide range of scenarios.

The concepts of the MOCCA component framework were published in [123], whereas the approach to parallel constructs was presented in [119].

Chapter 7

Interoperability Issues

In order to be usable in real-world scenarios, the component environment must be interoperable with other existing solutions and technologies. This chapter begins with a discussion on the issues of interoperability with other component models and their implementation on the example of the Grid Component Model. This is followed by a description of a solution for integrating CCA components developed in multiple programming languages with the support of the Babel tool. Finally, a brief discussion of interoperability with other systems using Web services is outlined.

7.1 Interoperability – introduction

The programming and execution environment which is the subject of the author's work has the ambitious goals to provide wide support for various types of scientific applications on the Grid. In realistic scenarios, however, one cannot assume that the system will operate in isolation. Conversely, the scientific applications and Grid infrastructures are, by nature, heterogeneous and rich, with plenty of other models, solutions and existing software. Therefore, interoperability, i.e. the ability to communicate and cooperate despite differences in the implementation language, the execution environment, or the model abstraction as defined in Section 2.5, is highly important.

First, the CCA is not the only component model which aims at supporting Grid applications: the Grid Component Model [40] proposed by CoreGRID and under standardization by ETSI (European Telecommunications Standards Institute) is a notable example; therefore interoperability with it must also be taken into consideration. The following section describes the author's approach to interoperability between GCM and CCA models, including a discussion on differences in component type system and invocation semantics. The proposed solution is based on the adapter pattern which, as we demonstrate, can be used to integrate components and component applications running in MOCCA and ProActive component frameworks.

The second issue, also very important for scientific applications, is the existence of many programming languages which can be used to develop components. Large

amounts of scientific code are written in Fortran and C++, which are not supported by many component frameworks, based mostly on Java. One of the advantages stemming from the decision for selecting the CCA model is the existence of direct support for multilingual components in the CCA standard. In this chapter, an outline of an approach to supporting Babel-based components with the MOCCA environment is provided, together with a report on the work done to integrate Babel-RMI extensions with RMIX, which is the underlying communication library of MOCCA.

It should be stressed that there the environment may have to operate with additional systems, some even unknown at present. To be ready for cooperation with these systems, the environment should include support for standard technologies, which can facilitate interfacing other remote software entities. The standard technology used most widely for distributed systems integration is the Web services set of standards and protocols. This chapter includes a discussion on interfacing the component framework with standard Web services. Combining dynamic deployment of components with a standard way of interacting with clients can yield a powerful and flexible environment for programming and execution of applications.

7.2 Interoperability with GCM

7.2.1 Introduction

The goal of this section is to address the problem of interoperability between GCM and CCA component models. Focus is on the base component model of GCM, namely Fractal [24], as it defines the fundamental properties of the components and their interactions. The discusion starts with an analysis of both models to identify similarities and differences between them. Subsequently, possible integration strategies are described, together with the solutions to the identified problems, such as typing system issues. A generic and framework-independent solution is proposed, which is based on the adapter (wrapper) design pattern. In order to validate the approach, the author has developed a prototype, which allows the ProActive (a GCM prototype) [11] and MOCCA (a CCA implementation) [123] frameworks to interoperate. The extensions to Fractal introduced in GCM, such as collective interfaces and autonomic controllers, are left for future work.

7.2.2 Overview and comparison of CCA and GCM

The CCA[9] specification is defined using the Scientific Interface Description Language (SIDL) [104], which specifies the core entities: components, ports and a framework. Ports are the external interfaces of a component and they must extend the `Port` interface. A component declares both its client and server interfaces called

uses and *provides* ports respectively. The framework is represented with respect to the component by the `Services` interface, used by the component to register its ports. This interface also defines a `getPort()` method which allows a component to obtain a reference to the uses port in order to invoke methods on this client interface. The external interface exposed by the framework to application developers is called the `BuilderService`. It provides methods for creating/destroying component instances and connecting/disconnecting their ports. Besides these core interfaces, CCA also specifies optional ports, such as component repository, connection event service, service registry and parameter ports, intended to facilitate interoperability between different frameworks.

Fractal is a *hierarchical* component model that provides *introspection* and *intercession*. It is easily *extensible* [24]. There are two kinds of components in Fractal: *primitive* components which are black boxes, and *composite* components that are composed of other components and can be used to build up yet other composites. Fractal enforces clear separation between functional and non-functional aspects: non-functional features are provided by *controllers*, and encapsulated in a *membrane*. This model provides reconfiguration (adding, removing, binding, and unbinding) of the functional content of composite components, in order to support adaptivity of component systems.

Fig. 7.1 presents an example of a composite Fractal component. Server interfaces are visible on the left, while client interfaces (uses ports) are on the right side of the component. The controllers are located in the membrane and shown on the upper bound of the component box. Inside the main (composite) component there are three components, one of which is, again, composite. The internal links show the binding between the composite component and its internal components: they show which interfaces are connected and which are exported to the outside. The ellipses inside the component boxes represent the component implementation (body), which in the case of the ProActive framework is an Active object.

The GCM is a component model targeted at Grid computing, which focuses on the following extensions to the base Fractal model:

- A *deployment* paradigm based on virtual nodes allowing to specify a logical deployment of a system, and a physical deployment separately.
- Support for *several communication patterns*. First, asynchronous method calls are considered as the default semantics, and other semantics, such as streaming and event-based communication may be supported. A major contribution of the GCM is to standardize multicast and gathercast interfaces that allow 1-to-n and n-to-1 communications.
- Support for *non-functional adaptivity and autonomicity*. The GCM specifies how to design non-functional aspects in a component-oriented way, and thus

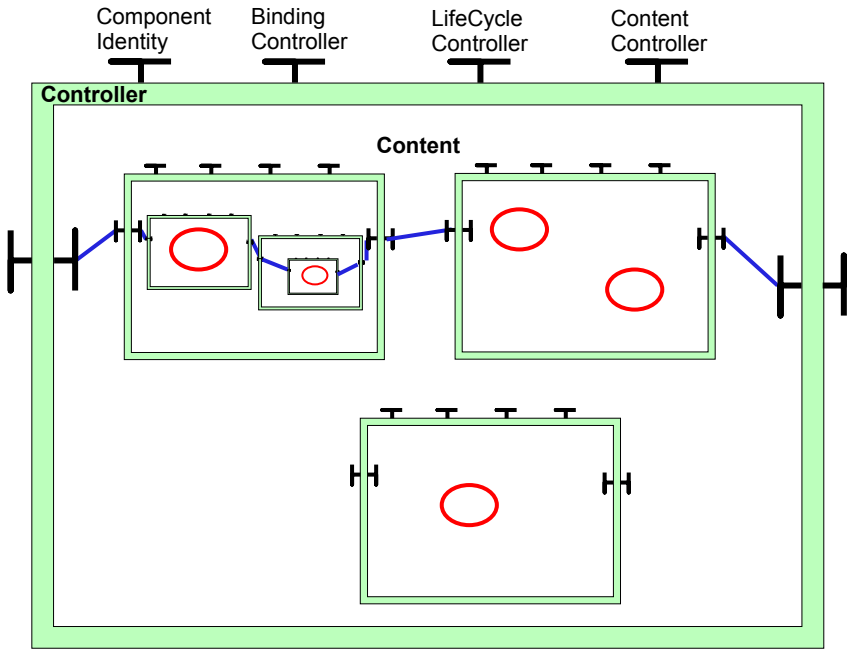


Figure 7.1: Example showing how composite components are modeled in Fractal. Ellipses represent component implementation.

allow the reconfiguration of non-functional features of a component system. Finally, a set of autonomic controllers is also standardized and they allow components to adapt themselves in a hierarchical and autonomous way.

Both CCA and Fractal component models enforce separation between the interface and its implementation, allowing composition of applications by connecting client and server ports of components, and providing some reflective capabilities.

The basic and obvious similarity is that the functional interfaces of components in both models are equivalent, e.g. when considering the Java implementation, both Fractal and CCA components are Java classes implementing their functional interfaces and some additional interfaces imposed by the specification. Interaction between components in both models is based on method invocation on the client interface which is connected to a server counterpart.

The first conceptual difference is the way the components (in both models) interact with the outside world. In CCA, a component is given an explicit reference to the framework, and the component itself has the "initiative" to actively inform the framework about its internals, i.e. ports (interfaces). On the other hand, the Fractal model assumes that the component plays a passive role in the introspection process and can reveal its internals on demand.

The second difference is the way the component interfaces are connected. In CCA the `BuilderService` is responsible for creating the connections and the framework manages them, while the component is only required to invoke the `getPort()` method to get a valid reference to the port prior to using its client interface. In Fractal, the connection is managed by the component, by implementing a `BindingController` interface.

`ContentController` in Fractal does not have a counterpart in CCA because CCA does not support composite components as explicitly as Fractal. Furthermore, there is no standard life cycle controller mechanism.

Although CCA does not distinguish non-functional interfaces (controllers) there are some standard ports, which are optional. One of them is is a `BasicParameterPort` which can be used to read and modify arbitrary properties of a component, analogously to the Fractal `Attribute` controller.

The mechanism of component creation is also different in both models. The method for creating instances in CCA is included in the `BuilderService` port, whereas Fractal defines the `Factory` interface for this purpose. In both cases, the creation mechanism may be implementation-specific, and depends on the actual framework.

Although there is no standard Architecture Description Language (ADL) for CCA components, the `BuilderService` provides all the required functionality to construct such a description. The Application Factory project defines an XML-based ADL for XCAT [106], whereas CCAFFEINE [9] defines its own scripting language for composing applications.

7.2.3 Overcoming typing and ADL issues

One of the main issues in this work is to deal with the fact that Fractal (and GCM) components have an immutable type (i.e. the set of exported interfaces cannot evolve dynamically) whereas CCA components can subscribe new ports to be exported at any time. More precisely, in CCA, each component can register a port at any moment, so there is no concept of component type. Conversely, in Fractal, other than for collection interfaces (which can be instantiated several times during the lifecycle of a component), the type of a component and the set of its interfaces is fixed upon its instantiation. The "static" typing of Fractal components can be used to verify the correctness of the bindings, according to interface types. The type of a Fractal component is usually defined in an ADL description. The following ways of solving the typing issue can be proposed:

1. Generate a Fractal component automatically upon instantiation of a CCA component, i.e. using only the port declared by the `setServices` method.

This allows building a Fractal component automatically, without any additional code (no ADL has to be specified) but prevents adding new ports following component initialization.

2. The programmer should specify the ADL for the CCA component. This requires additional manual effort, but no set of interfaces has to be automatically inferred. One of the main advantages of this approach is that some ports provided during component lifetime can be specified as Fractal *optional* interfaces.

3. An improvement of the previous approach consists in generating the ADL specification upon a CCA component instantiation (not necessarily the real one) and then reusing the ADL inferred in scenario 2 above. The user may then modify the generated ADL (to add some of the ports that will be provided during component lifetime).

4. One can also generate an ADL from the available CCA description (e.g. as SIDL [104]). The CCA script language (used by frameworks, but not standardized) may be reused to declare which ports of the CCA component / assembly should be exported.

It has been decided to choose the second approach as it seems more general, enables very good understanding of the differences between CCA and Fractal, and is centered on the interaction between the two frameworks. Moreover, it can be automated later on, with solutions 3 and 4.

All the aforementioned approaches require a mapping between exported CCA ports and GCM interfaces. More precisely, CCA ports are identified by the component name and port name, and this must be mapped to Fractal interfaces defined in the ADL. In other words, there is a need to define a bijection between CCA ports (i.e., component name + port name) and Fractal interfaces, as defined in the ADL.

7.2.4 Integration strategies

It becomes useful to separate CCA integration with GCM into two approaches: encapsulation of a single CCA component into a GCM component (Section 7.2.5) and wrapping a complete CCA system, consisting of several CCA components as a composite GCM component (Section 7.2.6).

Along the lifetime of a CCA-Fractal composition, the integration framework must support: (a) communication from the Fractal component system to the CCA system; (b) communication from the CCA system to Fractal components; (c) plugging or unplugging Fractal interfaces to the CCA system (both on the client and on the server side); (d) exporting new CCA ports, if supported (see Section 7.2.3).

Additionally, the solution should be as general as possible, i.e. generally independent of the CCA framework implementation.

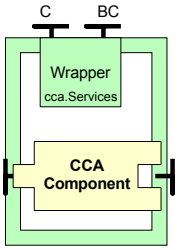


Figure 7.2: Integration of a single CCA component into a Fractal one

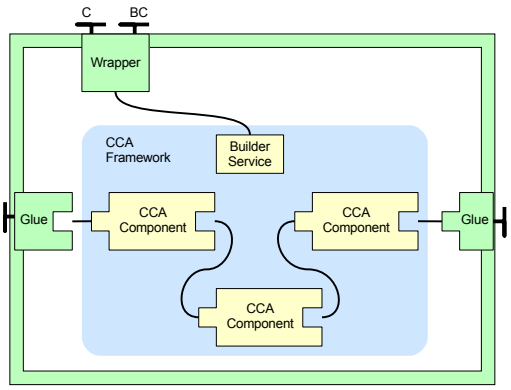


Figure 7.3: Interoperability between CCA and Fractal components

7.2.5 Simple integration

The discussion begins with a simple case: how to encapsulate a single CCA component as a Fractal component?

The proposed solution enables the creation (instantiation) of a CCA component as a primitive Fractal component in a single address space. It relies on a wrapper that encapsulates a CCA component, and exposes a `cca.Services` interface for that component (see Fig. 7.2). Prior to instantiation we should know the type of the component, in order to define the associated Fractal type. This may be obtained from an ADL description.

At runtime, the wrapper stores the references to the bound Fractal interfaces and passes them to the `getPort()` method of the CCA component as needed. All communication is performed by the Fractal framework (no need for any operating CCA framework at all – the wrapper constitutes a mini-framework for that component).

7.2.6 Framework interoperability

In this case, CCA components are created in their own framework and are connected to Fractal components running in their framework.

Complete interoperability between two frameworks requires instantiation of a whole CCA assembly, and the ability to interact with it from a Fractal framework as if it were a Fractal composite component. In this case, there is a CCA component or a set of CCA components which are created and connected among themselves by a CCA framework (e.g. MOCCA). Thus, the component assembly can be wrapped as a Fractal component in such a way that it can be connected to other Fractal components.

The proposed solution is based on a wrapper which adds a Membrane to a CCA assembly. The wrapper should interact with the CCA framework only via the `BuilderService` external interface (obtained in a framework-dependent manner). The wrapper is given a mapping between CCA system ports and external Fractal ports as discussed in Section 7.2.3. Using this information, it creates GluePorts as CCA components (using the `BuilderService` for each of the exported ports). The implementation of a GluePort is framework-specific, and translates Fractal invocations to CCA invocations and the other way around. The GluePorts expose Fractal interfaces to the outside world, and they can be connected (bound) to other Fractal components using the `BindingController` and `Component` interfaces of the wrapper. The wrapper uses the `BuilderService` to connect exported CCA ports to corresponding GluePorts using the CCA framework, so communication between the CCA component assembly and GluePorts is handled by the CCA framework.

In other words, the Wrapper component is both a CCA and a Fractal component. Although Fig. 7.3 shows the CCA system "inside" the wrapper, it is also possible to see the Fractal system from the CCA perspective as a "wrapped" one. Thus, the solution is symmetric.

7.2.7 Implementation - ProActive and MOCCA

In order to verify the proposed solution a prototype was developed using Java-based ProActive and MOCCA implementations.

Integration of a single component was realized as planned in Section 7.2.5. A wrapper which encapsulates a CCA component and exposes the `Services` interface to a CCA component is created by the Fractal framework. The wrapper instantiates the CCA component as a local object and invokes `setServices (this)` on a CCA component, passing a reference to itself. The CCA component registers its uses and provides ports, and consequently the wrapper can create direct (local) bindings to exported CCA ports.

In the framework interoperability scenario we assume that there are CCA components running in a framework and connected using a mechanism specific to this framework (e.g. a script, or Java API), forming the existing CCA assembly. Fig. 7.4 shows an example of wrapping an assembly of three CCA components which provides one port of type `A` and uses one port of type `B`. The scenario consists of the following steps:

1. The Fractal framework creates a `CompositeWrapper` component.

2. The wrapper implements a `CCAController` which is used to pass a description of the CCA assembly to the wrapper. This description includes all parameters required to connect external ports of the assembly.

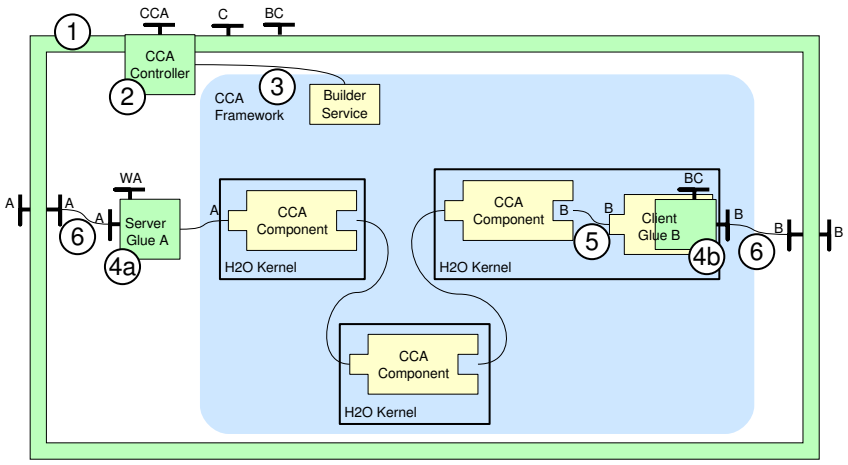


Figure 7.4: Wrapping an assembly of CCA components running in the MOCCA framework as a composite Fractal/ProActive component

3. A reference to `BuilderService` is returned by the framework-specific bootstrap method. In the case of MOCCA, the reference is obtained from the URI to Builder pluglet.

4. The Wrapper Component type is obtained from an ADL or Fractal API invocations. Provided with a mapping from CCA ports to Fractal interfaces (Sec. 7.2.3), the wrapper creates the GluePorts:

   (a) For each Provides port of the wrapped CCA assembly one ServerGlue port is created. It is created as a primitive Fractal component with one server interface and has one attribute controller called WrapperAttributes, which is immediately used to pass a reference to the corresponding CCA provides port (see e.g. `ServerGlue A` on the Fig. 7.4). The ServerGlue component contains MOCCA client code which delegates the method invocation to the wrapped component.

   (b) For each Uses port of the wrapped system one ClientGlue is created: it is a primitive one, becoming *at the same time* the Fractal and CCA component. It is instantiated in H2O kernel (a container for MOCCA) and upon creation it launches a ProActive runtime to expose the BindingController (BC). Consequently, ClientGlue can be connected to CCA components on its server side and to Fractal interfaces on the client side (see `ClientGlue B` in Fig. 7.4).

5. The wrapper uses the `BuilderService` to connect the exported CCA uses ports to corresponding GluePorts.

6. CCAController connects all Glue ports to the composite Wrapper using standard Fractal bindings.

7. The Fractal BindingController of a composite wrapper may be used to connect exported ports to other interfaces of the Fractal application.

It should be noted that both Client and Server Glue components are conceptually symmetric and their role is to translate invocations from one framework to the other. It was an implementation-phase decision to create Server Glue as ProActive component which includes MOCCA code, whereas Client Glue is created as a MOCCA component with an "embedded" ProActive one (Fig. 7.4).

7.2.8 Conclusions

Analysis of CCA and GCM component models shows that despite some differences, it is feasible to integrate components from one model into another framework, as well as to create glue code which enables inter-framework interoperability. The prototype functionality has been verified with a number of examples, including a non-trivial application (simulation of gold cluster formation[119], see also Section 9.9), and integrated with the ProActive library.

One conclusion is that if the properties of two different component models can be well understood, then the generation of wrappers and glue code bridging two different component frameworks can be generic and thus automated. This shows the general benefit of using a component model: integration of components developed in different standards can be achieved on the component framework level, rather then on the component level. Such a generic solution for two component models can be subsequently applied to all components of these models.

The presented approach resembles the one adopted in SciRun2 [146] with Bridge components acting like GluePort components. However, the proposed solution avoids introducing the notion of a new (meta) component model and allows components running in their native frameworks to interoperate (i.e. not requiring an additional framework).

7.3 Multilanguage interoperability

The second type of interoperability which is considered important from the scientific applications' point of view is the ability to combine components developed in multiple programming languages. The reason for that is the existence of many native scientific libraries, implemented years ago but still successfully used by scientists all over the world. One reason for their popularity is their efficiency that comes, for example, with a well-optimized implementation in the Fortran language. It is

well recognized that multilanguage interoperability is difficult in the general case, and would require manual or semi-automatic generation of wrappers or adapters. However, for a specific programming model, with specific constraints and with well-defined APIs, this goal is easier to achieve. One example is MPI, where Java wrappers exist. Similarly, the CCA model with constraints imposed by SIDL and with standard mechanisms of connecting component ports makes language interoperability possible.

The natural solution for an environment based on the CCA model is to use the capabilities that Babel offers. This basically involves writing a component implementation in any supported language (C, C++, Fortran 77 or 90, Java, Python) in conformance with the SIDL interface definition. Subsequently, the component may be shipped as a dynamically-loaded .so library file accompanied by its SIDL interface definition, and the framework should be able to load it and connect it to other components. To achieve interoperability, Babel defines its internal object representation (IOR) which is a structure of function pointers written in C. Each language binding creates a stub and skeleton code (automatically generated from SIDL), to allow interaction of the specific language program with the IOR. In the case of Java, the Java Native Interface (JNI) is used. The Babel runtime library is required to interact with all language endpoints in the case of a single process, and recent extensions of Babel, called Babel-RMI, allow plugging in communication libraries to enable remote method invocations on Babel objects.

This section describes an approach to multilanguage interoperability in the component environment. It is divided into two stages: the first step involves integrating Babel-RMI with the RMIX communication library which is used by the H2O and MOCCA framework. The next step is the integration of such a Babel-RMIX solution into the MOCCA framework. The following subsections describe in detail how the first step is realized. Integration concepts are outlined later on. The actual implementation of the final solution is out of the scope of this thesis and it is left for future work.

7.3.1 Babel background

To understand the technical details of the proposed approach, it is first necessary to introduce the main concepts of the Babel system, which forms the core of the solution. Babel provides, as its main functionality, a mechanism for interaction between multiple programming languages within a single process. This is achieved through the introduction of the Intermediate Object Representation (IOR) which is a special structure written in C, which includes a list of function pointers referencing the object implementation. As shown in Fig 7.5, a client program (written e.g .in Java) invokes a method on a *stub* object, which delegates the invocation via IOR to the *skeleton* object, responsible in turn for invoking the actual implementation

(written e.g. in C++).

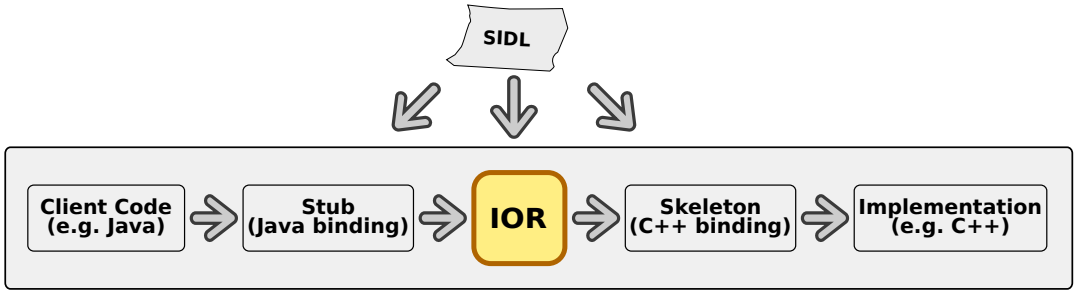


Figure 7.5: Babel operation in single process

The interfaces of objects which are accessible through Babel have to be described using the Scientific Interface Definition Language (SIDL). It is similar to CORBA IDL [138], however it supports such scientific data types as multidimensional arrays and complex numbers. Babel includes a SIDL parser and code generator, which automatically generates IOR, client-side stubs, server-side skeletons, template implementation source files and scripts for compilation (makefiles, etc.)

In the case of remote client and server objects, Babel offers an extension, called Babel-RMI. It allows plugging in multiple communication libraries, which may support various communication protocols. Its operation is depicted in Fig. 7.6 which shows a client accessing a remote server (library) wrapped as a Babel object. First, to be remotely accessible, the server object has to be exported using a babel Runtime. Exported objects are identified using URIs, e.g. `rmix://host.example.domain/rmi_server/ObjectId` represents an object accessible using the `RMIX` provider, on a host `host.example.domain`, an instance of which is identified with `ObjectId`.

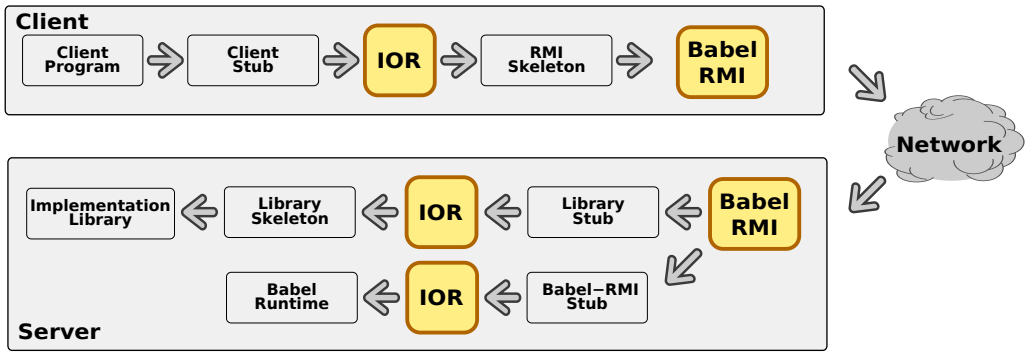


Figure 7.6: Client-server interaction with Babel-RMI

The first step for the client is to obtain a remote reference to the server object using the provided URI. The Babel runtime parses the address and passes the call

to the underlying RMI library, which deals with the transport of data through the network. On the library side, RMI deserializes the data and uses the Babel server-side implementation to interact with the actual object through IOR. Communication is based on using predefined interfaces that are shipped with the Babel framework.

Babel is designed as an extensible framework, thus it allows plugging in third-party communication libraries as Babel-RMI providers. Babel offers a set of interfaces to be implemented by library providers, grouped in `sidl.io` and `sidl.rmi` packages:

- `Serializer` and `Deserializer` – responsible for marshalling/unmarshalling all datatypes supported by SIDL;
- `Serializable` – to be implemented by objects which require specialized marshalling;
- `InstanceHandle` – enabling to connect to remote object;
- `Invocation` – extends the `Serializer` and performs actual method invocation;
- `Response` – performs deserialization on the client side following successful remote invocation;
- `Call` and `Return` – perform (de)serialization on the server side;
- `ServerInfo` – enables exporting Babel objects remotely and produces a URI which can be used as a remote reference;
- `ProtocolFactory` – manages existing RMI implementations, allows registering and deleting them;
- `InstanceRegistry` – responsible for generating unique identifiers of object instances;
- `ServerRegistry` – allows obtaining `ServerInfo` of registered RMI servers.

In addition to these communication-specific interfaces, Babel offers a pluggable mechanism to manage dynamic library loading. Babel internally manages loading of shared libraries and symbol tables within them. It is also possible to implement a new library `Finder` interface, to search for libraries in a specific place. This can be used by the component container to plug in specific loading mechanisms.

7.3.2 Concept of integration of Babel with RMIX and MOCCA

Introducing support for Babel within the MOCCA component environment was divided into two stages:

- Integration of Babel with RMIX
- Integration of Babel with MOCCA

RMIX – Babel

Integration of Babel with RMIX is a natural solution, since RMIX is a communication library for H2O which is the basic platform for the MOCCA component environment. As Babel-RMI extensions were published shortly after the beginning of the author's work with MOCCA, it was decided to investigate the possibility of integrating these two solutions.

The potential advantages of this solution are more generic and can be used outside the MOCCA and H2O environments, since RMIX can be used as a standalone communications library. Advantages include support for multiple underlying protocols (such as JRMP, SOAP, RPC) which are supported by RMIX. Moreover, such features of RMIX as transport tunnelling, compression and security with the usage of SSL are directly supported. As RMIX allows the use of alternative network layers, such as JXTA instead of IP (see Chapter 8), RMIX-enabled Babel objects can communicate using e.g. Peer-to-Peer overlay networks. There are obviously possible drawbacks of RMIX which, as a Java-based library, cannot achieve performance equivalent to that of dedicated libraries which use a communication layer directly on the operating system level and implement fast serialization and deserialization functions.

The design of the Babel-RMIX integration assumes the usage of the Babel-RMI extensions and plugging in RMIX as a communication library provider. A diagram representing this design is shown in Fig. 7.7. All Babel-RMI interfaces should be provided by the RMIX implementation, including serializers, invocation handles on both client and server side, factories and registries. As RMIX is a Java-based library, all classes use Java bindings to Babel. The URLs to Babel objects accessible using the RMIX provider use the `rmix` protocol prefix, as in: `rmix://host.example.domain/rmi_server/ObjectId`.

MOCCA – Babel

Integration of MOCCA with Babel requires tighter coupling between these two systems. As described in Chapter 6, the base component environment, called

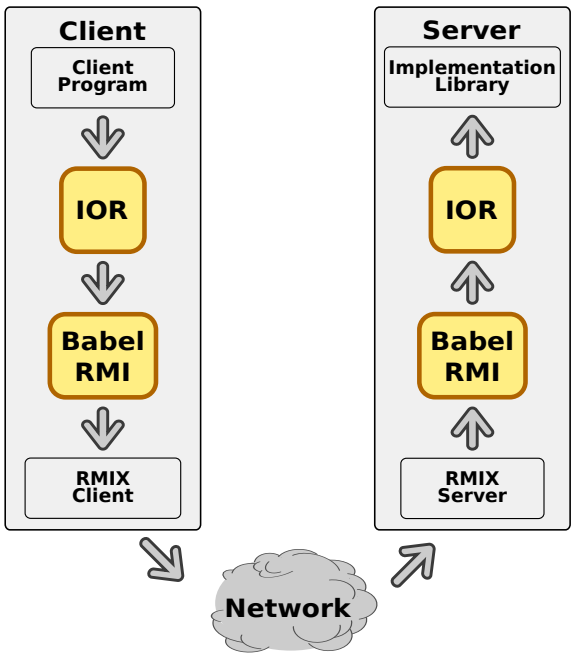


Figure 7.7: Design of Babel-RMIX integration

MOCCA_Light, is a pure Java-based CCA implementation built on top of H2O. It supports dynamic component deployment to the H2O kernel and uses RMIX for communication. The most ambitious goal of integration with Babel would be to add into MOCCA full support for *binary* CCA components, possibly implemented in and compatible with different Babel-compliant CCA frameworks, such as CCAFFEINE from the CCA Toolkit.

To integrate with Babel, two important enhancements need to be introduced in comparison with MOCCA_Light:

- Adding a SIDL-compliant interface between components and the framework;
- Providing a mechanism for deployment of binary (native, i.e. non-Java) components into the container.

The proposed architecture of the first enhancement is depicted in Fig. 7.8. A Babel-compliant component communicates with the external world via its SIDL-compatible interfaces: *uses* and *provides* ports. It needs to obtain a reference to the CCA framework using the `cca.Services` interface, in the same way as was shown in Fig. 6.3. In this case, the MOCCA implementation of the client-side Services interface should use Babel-RMI library, and, following a successful connection, should return a stub to remote *provides* port. On the server side, the *provides* port should

be exported by Babel-RMI. It is important to note that in such a scenario communication is handled directly by Babel-RMI, independently from the actual RMI provider library, so e.g. built-in Babel-RMI could be used. On the other hand, H2O mechanisms should be used to perform actual management of components, such as connections and passing references to the remote ports.

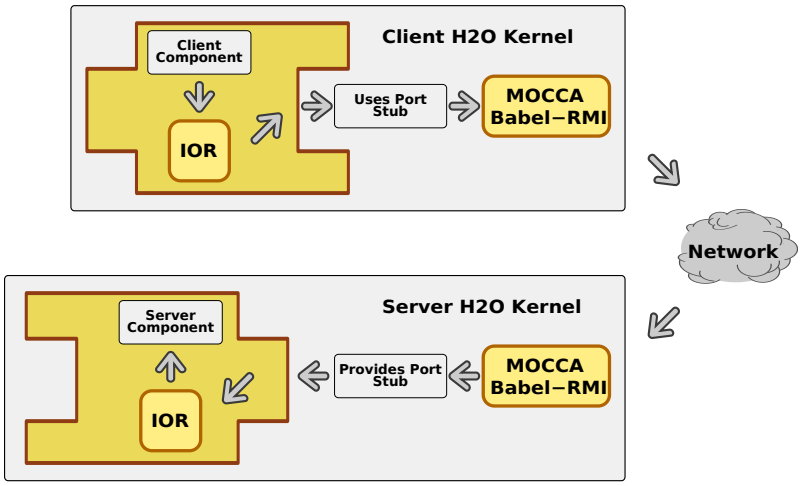


Figure 7.8: Design of Babel – MOCCA integration

The second enhancement requires a mechanism for deploying a Babel-compliant component into the H2O kernel. This is not a trivial issue, since such components are no longer platform-independent as was in the case with pure-Java ones. Babel components are packaged as dynamically-loaded library (DLL) files, depending on the target platform (e.g. .so ELF files on Linux). One solution would be to restrict the usage of components to a specific platform, such as Linux on x86 architectures, which is dominant in production Grids. Another option calls for pre-packaging of components for multiple platforms and bundling them together. Such a mechanism is supported by H2O which uses the PVM_ARCH property to declare platform type. Another solution would be to package components in their source format and compile them at deployment time, but this is a cumbersome procedure and not yet standardized in CCA, although mechanisms to facilitate building of CCA components are under development within Bocca [51].

Another issue related to the deployment of native components comes are issues with loading shared libraries into the Java Virtual Machine (JVM). Java bindings to Babel rely on the usage of the Java-to Native Interface (JNI) which provides support for running native code inside the JVM. However, this solution has many limitations, including in particular the loss of Java security features (native code may use pointers to access arbitrary regions of the JVM process, hence erroneous or malicious code can crash the JVM). It would be possible to allow deployment

of native code into the H2O kernel JVM in the case of trusted and e.g. digitally-signed components only, but in a general case, such a solution is unacceptable for the Java-based container. What remains as a possible solution would be to create a new process for each component or a set of components belonging to a single user or application. Subsequently, operating system mechanisms could provide stable isolation between the components and not cause container instability. Additionally, other restrictions can be made, e.g. by running each component with a different UNIX user ID, or even using virtual machines to provide lower-level sandboxing.

7.3.3 Implementation status and conclusions

In the course of the presented work, it was decided to focus on the development of the first step of the solution, namely integration of Babel with RMIX. This work included designing the classes which implement the Babel-RMI interfaces as well as underlying communication calls. The design concept was based on the introduction of a special class, called *RMIXBean*, which was used as a packet containing all serialized method parameters and was transmitted over the network. The results of performance tests of the implementation [121, 83][1] show that using Java for serialization can be orders of magnitude slower than low-level C-based Babel-RMI implementation; however on lower-bandwidth networks serialization does not become a bottleneck (compare performance tests of MOCCA and XCAT in Chapter 9).

Implementation of full support for Babel-based components, as outlined in the preceding section, is out of the scope of this dissertation. Experience with implementing Babel-RMI and some preliminary experiments conducted with the Babel and H2O code, indicate that such integration may be feasible. However, even though tackling the issues of deployment and interfacing between native code and Java code can be time-consuming and fraught with errors, it remains an important goal, relevant for real scientific applications, which may be written in multiple programming languages.

7.4 Interoperability using Web services

Web services, a set of XML-based technologies including the SOAP [189] protocol and WSDL [188] interface description language, have become a standard solution for achieving interoperability in Internet-wide distributed systems. They gained popularity due to programming and operating system independence, by relying on the exchange of XML-based text messages. As discussed in Chapter 2, Web services do not provide such mechanisms as composition in space (i.e. direct connections, as

[1] A prototype implementation of the concepts described in Section7.3 was performed by Daniel Harężlak, co-author of the paper [121].

in the component models) and exclude deployment from their associated standards. Nevertheless, a communication protocol and service description standard can provide a valuable and convenient way of providing an external standardized interface to the component framework.

Web services have been used in other component frameworks to provide interoperability. One example is XCAT [106], which is a distributed CCA framework where Web services are used as a main communications layer, applying the custom XSOAP [75] library. An advantage of this approach is the possibility of ensuring compliance with such standards as OGSI and WSRF, however restricting oneself to one communication protocol can be considered a limitation. Another example is the ProActive framework [11], where the *active objects* which form the core of the programming model can be exported as Web services. ProActive provides a method to export active objects using proxies deployed to the Apache SOAP engine running in a standalone Tomcat servlet container. Another recent technology which combines Web Services and components is the Service Component Architecture (SCA) [14]. It enables spatial composition of components implemented in multiple languages, including Java, scripts and BPEL, communicating using such protocols as RMI, SOAP, JSON-RPC, JMS. SCA is quite new, but it shows promise since it tries to combine the advantages of both component and service-oriented models.

As can be seen in the above examples, the ability to interoperate with other Web service-based systems is an important feature of a realistic programming environment. This section describes an approach to achieving such interoperability in the presented component environment, using an exporting mechanism.

MOCCA, which is the core of the programming environment, is based on the H2O platform, which, among other features, offers a communication substrate called RMIX. RMIX is a miltiprotocol communication library, supporting SOAP as one of its protocols, using embedded XSOAP or the Apache Axis engine. In principle, it is possible to communicate with it from any SOAP client. However, H2O does not provide WSDL descriptions of remote interfaces, which is a drawback when it comes to integration with third-party clients. Therefore, one of the requirements for the exporting mechanism is to expose a given *provides* port of a component as a Web service with a proper WSDL description.

The proposed solution is similar to the concept outlined in the paper describing an approach to the interoperability of H2O with the OGSI standard [164]. An *exporter component* is introduced, which acts as a Web service proxy to the exported port. The details of the exporter component are shown in Fig. 7.9. Exporter is a MOCCA component, which, on the one hand, is a client for the exported port, and on the other contains an embedded SOAP engine which publishes the port in a standard way. Therefore, non-Java clients, using e.g. the Perl SOAP::Lite library can connect to the exported port. The exporter component can be deployed either

on the same H2O kernel as the component of the exported port, or on a separate machine, and can then serve as a gateway handling multiple ports. Additionally, the MOCCA builder service needs to be extended to support a method for exporting a given port of a component as a Web service.

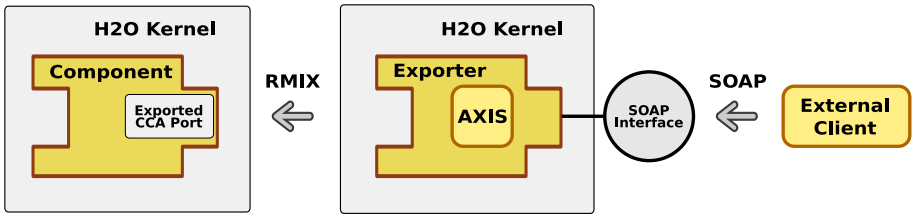


Figure 7.9: Design of the *Exporter* component which exposes provides ports of the MOCCA components as Web services.

Initial experiments with prototyping of the exporter component using Apache Axis as a SOAP engine and an embedded Jetty Servlet container have been performed. The conclusion is that invocations which involve primitive types, such as strings, numbers, arrays, etc., are quite straightforward to implement. Handling complex types, such as arbitrary Java classes, would require definition of specific data bindings and XML mappings, which is, in general, a more complex issue requiring more development effort.

7.5 Conclusions

This chapter provided a discussion on the interoperability issues related to the component programming and execution environment for scientific applications. Interoperability by nature requires compliance with standards which define common interfaces, protocols, data formats, etc. In the area of distributed programming environments for scientific applications, the author identified three standards which were of importance from the point of view of interoperability.

The first one is the Grid Component Model proposed by the CoreGRID project and under standardization by ETSI. Comparison between CCA and GCM and a study of the prototype interoperability layer between MOCCA and ProActive implementations demonstrate that although the core models contain some fundamental differences, it is possible to provide interoperability solutions with reasonable restrictions.

The second standard which was considered is called SIDL, an interface description language for scientific components, supported by CCA and the Babel toolkit. Interoperability with Babel-compliant components was outlined and, as part of the

solution, Babel-RMI was integrated with the RMIX communication library, which is the underlying transport solution for the MOCCA component environment.

The third standard which the environment should be interoperable with is Web services, strongly supported in the scope of Grid and scientific computing by such initiatives as OGSA [53]. This requirement is partly satisfied by the RMIX communication library which can use the SOAP protocol, whereas experiments with an exporter component suggest that adding WSDL support is feasible.

The solutions presented in this chapter, although applied to a specific component model and framework, can be considered generic and thus applicable to other component models and frameworks. This is possible due to reliance on standard concepts and design patterns, such as adapters and proxies (exporters). The proposed solutions can also serve as a base for higher-level interoperability, using e.g. Semantic Web standards such as ontologies (which is, however, out of the scope of this thesis).

The GCM-CCA interoperability study, as described in this chapter, was published in [118]. Details of the work on Babel integration were presented in [121, 83].

Chapter 8

Deployment on Production Grids

The vision of component- or service-oriented Grid infrastructures is currently in its early development stage, while the production Grid infrastructures offer only simple batch job processing, with communication between nodes often limited by firewalls and private networks. This chapter describes how a component-based application can be executed on a production Grid infrastructure through dynamic deployment of a pool of component containers. The communication layer can then be extended with the introduction of a peer-to-peer overlay network, such as JXTA, to enable direct communication between all computing nodes.

8.1 Issues with production infrastructures

The component-based programming and execution environment discussed in this thesis assumes the existence of some low-level computing infrastructure, where the application components can be deployed. Such an infrastructure, in the specific case of the MOCCA framework, requires the existence of component containers, namely H2O kernels to be installed on all of resource provider machines. Although this would be an ideal scenario and can be set up in experimental testbeds, such as e.g. Grid'5000, it is not the case with existing production Grid infrastructures.

The majority of existing production Grid infrastructures, such as EGEE, DEISA and others enumerated in Section 1.2.2, are based on the concept of batch job processing and do not support higher-level programming models. Their advantage, on the other hand, is that they are generic, i.e. allow running arbitrary executable programs supplied by the user following successful submission of the computing job and depending on the available resources, queue lengths, etc.

Another problem with infrastructures such as EGEE, are obstacles for distributed computing applications, since network connectivity between the nodes of computing clusters is restricted. This is due to the fact that the computing nodes of clusters comprising Grid sites are typically hidden in private networks (behind firewalls and NATs), thus enabling only outbound Internet connectivity. This is a major problem when running parallel or distributed applications which go beyond simple processing

of independent tasks, since it becomes impossible to open a direct connection from one cluster node to another.

The goal of Section 8.2 is to describe an approach to dealing with the first issue by providing a *method for deployment of component containers on the production Grid infrastructure*. Such a user-managed pool of containers may span multiple Grid sites and also multiple infrastructures, e.g. EGEE and NGS, as well as other resources, e.g. local PBS-administered clusters. Provided a specific resource discovery system, the created pool of resources can be seen as a user-centric virtualization layer, thus *reducing* the problem of deploying applications onto the infrastructure to the problem of deploying components onto a set of containers.

To deal with the second issue, namely communication restrictions, it is possible to reuse concepts known from peer-to-peer (P2P) networks. In such a system, communicating peers can create an overlay over a physical network, where the relay peers (also called super-peers) can act as routers which enable communication between all peers, including those in private networks. Section 8.3 demonstrates how the JXTA [99] P2P framework can be used to provide communication between the computing nodes in a component-based application. The solution is based on replacing the standard TCP socket in the RMIX communication library with JXTA sockets which provide a uniform transport layer over the JXTA P2P overlay network.

8.2 Deployment of component containers on Grids

There are situations where a user has access to multiple virtual organizations built upon different Grid infrastructures, as well as local clusters or single machines that are not part of Grids. Such a user may wish to run a large-scale application on all the available resources simultaneously. This should be possible without the need to set up a new virtual organization, which usually requires Grid middleware interoperability and imposes a huge administrative overhead. Instead, the *user-centric* scenario should allow for ad-hoc and transient collaborations formed spontaneously across Grid systems.

The issue is to devise an efficient method for user-centric aggregation of the computing power of such distributed and highly heterogeneous resources, which may have different access rules, application setup and execution mechanisms, and may belong to many virtual organizations. The aggregation method should hide the complexity of the underlying infrastructure and provide a virtualized layer for executing the application.

In this section the author proposes solutions to this problem which combine the component programming model for distributed applications with methods for aggregation of available infrastructures to gather computational resources across the boundaries of Grid and cluster systems. The described approach is user-centric and

avoids the need for involvement and effort on the part of Grid system administrators.

The proposed approach was validated by the author with experiments on a computationally-intensive minimization application using the Common Component Architecture (CCA) [9] model implemented in MOCCA [123] framework based on the H2O platform [108] (see Section 9.7).

8.2.1 Aggregation of computer resources – related work

It can be observed that great enthusiasm for building Grid systems has led to the development of many middleware solutions and a diversity of installations across countries. As examples, one can point to such projects as the European DataGrid, CrossGrid, Unicore, GridLab and EGEE, the overview of which can be found e.g. in [76]. Unfortunately, even if the user has access to many of these systems, they are not interoperable, which makes it difficult to use their resources in an aggregated way (see also Section 1.2.2).

There are several initiatives targeting Grid interoperability (e.g. projects such as GLUE [69] and UniGridS [180]), but their goal is first of all integration of middleware and infrastructure at the organizational level and they require additional work on commonality issues concerning security, information systems and policies. Unification of various cluster resource management systems was the initial motivation behind the development of Globus GRAM, and hence this solution is also organization-centric.

To solve the interoperability problems, some solutions conceal middleware heterogeneity under an additional layer; an example is GAT [5], which enables interaction with multiple underlying Grid infrastructures. Another project, related to the user-centric harnessing of compute resources, is DIANE [131], which is a master-worker distributed analysis environment enabling efficient utilization of EGEE/LCG Grid. These solutions, however, do not constitute a higher-level programming model, as they are focused on simple job processing.

8.2.2 Aggregation of resources

Assuming that the user builds applications with a component-based programming model, a need arises for the development of a *method* of aggregating computing resources for application deployment. An important assumption is that there is a need for a solution to a scenario where a single user may have access to heterogeneous resources, with different access methods, possibly distributed across multiple Grid infrastructures which run on different middleware platforms and belong to separate virtual organizations. This section demonstrates how H2O middleware may be used for creating a virtual user resource pool, which can then be applied to deploying

and executing applications on the infrastructure *without the need for system administrators' involvement*. The following subsections describe the proposed approach to:

- setting up the user's virtual resource pool,
- executing the application on the resulting infrastructure.

The goal of these two stages is equivalent to the component deployment process, as described in Section 2.4. Separating these stages enables *hiding infrastructure heterogeneity* in the first step, following which the deployment and execution of the actual application becomes possible.

8.2.3 Infrastructure setup

The proposed approach to creating the inter-Grid, user-centric virtualization layer involves a unified fabric of (transient) H2O kernels spawned dynamically on Grid-enabled resources. For standalone machines this may be done directly, using SSH, whereas on a cluster or on remote Grid resources there is a need *to submit an H2O kernel as a batch computing job*. When the kernel is started, the resource becomes a configurable element of the user's virtual pool (see Fig. 8.1).

In a Grid infrastructure with an automatic resource broker the location of the computing node on the remote site where the kernel is running may be *unknown* to the user in advance. To solve this problem, a simple *discovery* mechanism to locate available kernels is proposed. As a discovery service, the author uses the Java Naming and Directory Interface (JNDI)-enabled Harness Distributed Naming Service (HDNS) system [72] operating on a centralized server; however, it is also possible to run multiple HDNS servers to provide fault tolerance. In addition to the discovery service, a simple scheduling mechanism has been conceived to automate the process of assigning components to available resources for deployment.

An important aspect of this approach is that it does not require any specific software pre-installation on the target computing nodes, which might involve huge administrative overhead, especially in the case of multiple Grid systems. Consequently, such an approach allows seamless installation of the required base software on the computing machines. This solution stems from the nature of the distribution of H2O software, which is designed to run out-of-the-box on any Java-enabled system. A preconfigured thin version of H2O distribution was prepared which also contains the MOCCA library and fits into a 4MB archive file. All that is required to install and run the H2O kernel, is to unpack the distribution and run the specified kernel executable. When accessing the target system using SSH or PBS on a local cluster the installation step can be done manually by the user. For a Grid infrastructure such as LCG [70], manual installation is not possible, because access to worker

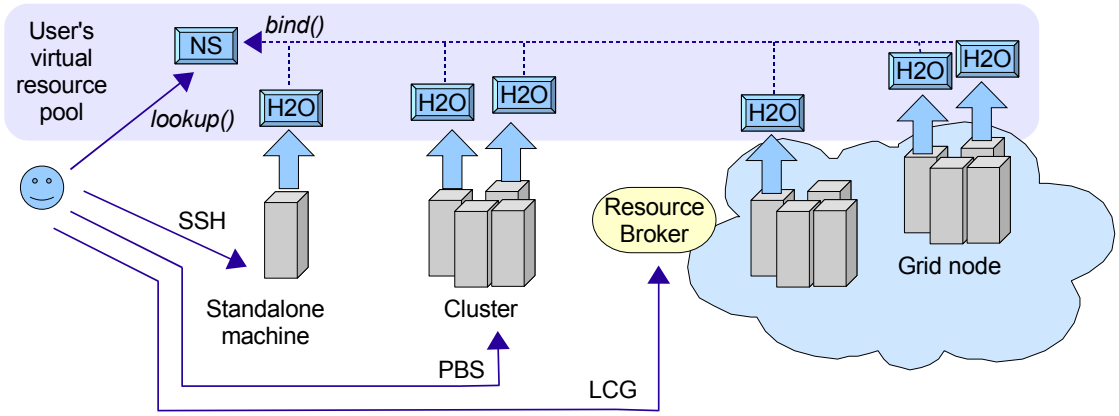


Figure 8.1: Setting up the user's virtual resource pool. The user starts the HDNS server ("NS"), and then spawns H2O kernels on any machines that are accessible. SSH may be used for standalone machines, the PBS queuing system – for local clusters and specific middleware, such as LCG with its Resource Broker – for Grid systems. Once the kernels are up, they are registered in NS and form the user's virtual pool of resources, available for deployment of the component application.

nodes of computing elements is available only through a job submission mechanism, i.e. via a Resource Broker, Globus and a local queuing system. Setting up the H2O kernel is then achieved by submitting a batch job which transfers this distribution on the working node, unpacks it and runs the kernel executable. The only requirement is that a Java virtual machine must be installed (JVM is present in most cases). It should be noted that if the specified version of JVM is not available on the Grid site, it will be possible to add the Java Runtime Environment installation as a first step of the job. If the need for transferring large files becomes an issue, it is possible to use the Grid replica management system to optimize file transfer time.

Another aspect which should be taken into account is that when an automatic Grid Resource Broker decides for a Grid node to run a batch job, the location of the H2O kernel started by this job may not be known to the user in advance. Locating such a kernel requires some form of discovery mechanism. For this purpose the HDNS naming service was used, which needs to be set up prior to running H2O kernels. Subsequently, when the preconfigured H2O kernel is started, it automatically loads the naming service client pluglet. Its role is to contact the discovery service and to register the new kernel on the server. The client contacts the HDNS server using JNDI API, which gives the possibility of future integration with other naming services, e.g. ones based on the LDAP protocol.

It is worth mentioning that the first step does not mandate the usage of a component model for application programming. Furthermore, both steps may also be performed by different actors, hence we can distinguish the resource pool provider from the application deployer or even from the final user, just like in H2O.

The security aspects should be also mentioned here. Current experiments were performed in an open system, which may potentially lead to unauthorized access to the pool of user's resources. However, it is possible to guarantee system security by means of an H2O kernel access policy. One of the options would be to preconfigure all kernels which belong to the user's pool in such a way that they accept only code signed by the specified user's digital certificate, consequently denying access to other parties. This would be in agreement with the proposed user-centric approach, since it is the user who sets up the H2O kernels and decides who can access a given pool of resources.

In the current experiments, management of the pool of H2O kernels has to be performed manually by the user. A set of scripts was prepared which may be used for submission of H2O kernels using the PBS system and Job Description Language (JDL) scripts for LCG middleware. A set of simple command-line tools may be used for querying the Name Service, testing the connectivity to remote kernels and performing shutdown of the pool. For monitoring of specific kernels, the H2O GUI can be used, yielding information on the state of deployed components.

8.2.4 Application deployment and execution

Once the required number of H2O kernels is running and registered in the discovery service, application execution may take place. It is worth noting that with a Grid infrastructure, whenever the pool of kernels is created, it will most probably contain different machines, chosen by the resource broker. In order to run the application on such a testbed, the user should prepare the components and the application description in the following way:

- each component of the application should be available as a JAR file published on the HTTP server,
- the script which creates the component instances connects them and finally triggers application execution.

Since the list of available H2O kernels is not known prior to runtime, there is a need to use some form of a *scheduler* responsible for automatic selection of locations for running the component instances. The scheduler contacts the discovery service to query about the available H2O kernels and selects them according to a specific policy. Whereas various approaches to component deployment are being researched

[39], for the purpose of these experiments the author has implemented a simple scheduler prototype which selects the kernels based on a *round-robin* policy. The scheduler may be invoked by the deployment script and the returned locations of kernels can then be passed to the *BuilderService* for creating single or multiple instances of components.

Once the components are instantiated on remote sites and their ports are connected, the script invokes the starter component and passes control to the application components. It is also worth mentioning that in order to receive application results, possibly even in an interactive way, it is preferable to run one of the components on the kernel, locally available to the user. This provides online access to application logs and output, whereas in order to obtain access to the files produced on Grid worker nodes, it is necessary to use additional Grid file transfer tools.

8.3 Communication using JXTA P2P overlay network

As discussed in Section 8.1, the existing production Grid infrastructures often impose certain restrictions when it comes to direct communication between computing nodes. This is caused mostly by the limitations of the current Internet based on the IPv4 protocol, which forces creating private networks due to the limited number of IP addresses. Computing nodes in clusters are then hidden behind NAT (Network Address Translation), which allows for outbound connectivity, but precludes inbound connections. Such limitations are often enforced by administrators for security reasons. All these solutions effectively hamper the possibilities of parallel-distributed computing spanning multiple Grid sites, and force the usage of specialized communication solutions based on proxy, gateway or tunnelling mechanisms.

This is actually the very lieu where peer-to-peer overlay networks may come as a solution. P2P systems, designed to harness peers gathered on the PCs of a multitude of users, were designed to provide communication (e.g. file sharing, telephony, etc.) between all of them, regardless of their location and without the requirement of public-IP Internet connectivity.

This section presents means of combining the H2O resource sharing platform with the JXTA P2P technology [99]. The goal is to demonstrate that synergizing the flexible and configurable resource sharing offered by H2O with the ability to accumulate and communicate the resources offered by JXTA, enables construction of a general-purpose P2P resource sharing platform, suitable for variable-scale metacomputing. This concept can be directly applied to the component-based programming environment to facilitate communication between the nodes of the production Grid infrastructures.

8.3.1 JXTA background

All the available peer-to-peer systems share the goal of rapidly finding resources and exchanging some information between users. Three main generations of P2P networks can be distinguished, namely simple filesharing systems such as Napster [32], fully decentralized frameworks such as Gnutella [151], and a third group based on the concept of superpeers or Distributed Hash Tables (DHT) [163]. For an overview of P2P computing and its interaction with Grids the author refers to [92].

JXTA is one of the most advanced P2P network implementations, considered to be the third generation P2P system, using the DHT concept. JXTA implements a discovery protocol, a transport protocol and peer group management. Developers can write custom services, making JXTA extendable.

JXTA conveys an abstraction of peers, which are basic entities and can be either a simple cell phone or a large server. It is important to note that peers in the JXTA network can collaborate by forming PeerGroups. Each group can use specific membership services restricting its members a set of peers which are able to fulfill certain requirements.

One of the most important features of the JXTA is its communication capability, evaluated in terms of usefulness in distributed computing [8]. Peers can create communication channels described as *pipes* that can connect entities behind firewalls or over different private networks. If the connection is made between peers in different private networks, dedicated proxy peers are used transparently (from the user's point of view). On the other hand, for pipes between peers in the same subnetwork, direct TCP connections are used to increase efficiency. In later versions of JXTA a familiar *sockets* abstraction was introduced on top of pipes.

Each resource in the JXTA network should be uniquely identifiable. This problem was solved in JXTA by the introduction of IDs that serve as a flat addressing type, independent from physical locations. For instance, a peer may connect to the network from home and disconnect after some time. Later on, when host reconnects to the network from a different physical location, JXTA will recognize this peer correctly.

8.3.2 Concept of a distributed computing framework using a peer-to-peer network

The problem with existing Grid infrastructures is that although they provide mechanisms to for sharing computing resources, support for resources which are behind firewalls or in private networks is very limited. There is a need for a distributed metacomputing system that will be able to act using a P2P network.

Figure 8.2 presents the proposed concept of a P2P computing system. As one

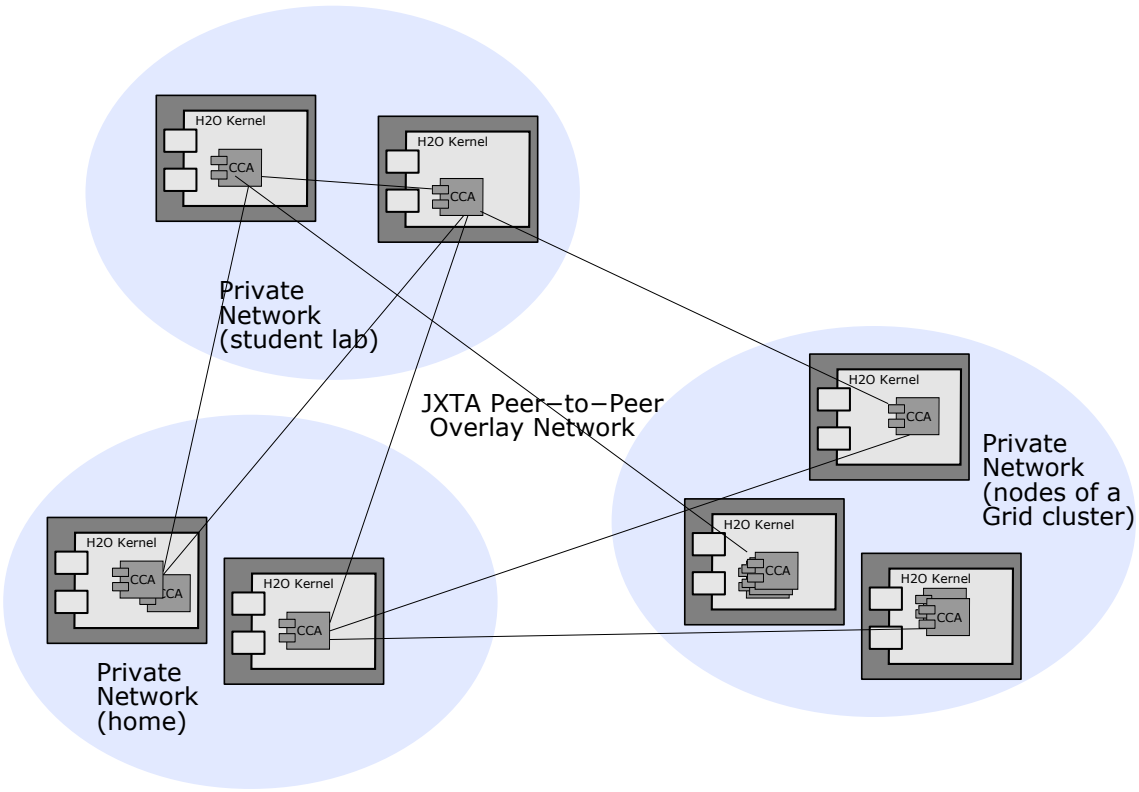


Figure 8.2: A concept of a P2P computing system

can notice, a P2P network spans multiple private or local area networks, including computers in student labs, home networks and nodes of Grid clusters. Resource owners from each private network make their resources available over the network. Additionally, they can decide to provide their resource over a specified virtual group. Resources can be accessed by anyone who joins the specified P2P group. One of the advantages of such a system comes from using flat addressing in P2P networks. Moreover, object addresses are independent from host IP addresses. This feature enables sharing a specified resource by using the same address independently from its location. Such features of P2P metacomputing frameworks create new possibilities for building global computing systems. The most important advantages are:

- Simplicity in building a computing network (no need for specialized configuration of routers or firewalls);
- Wider computing network (users from private networks can share resources);
- Clients can use the same resource independently from its location;

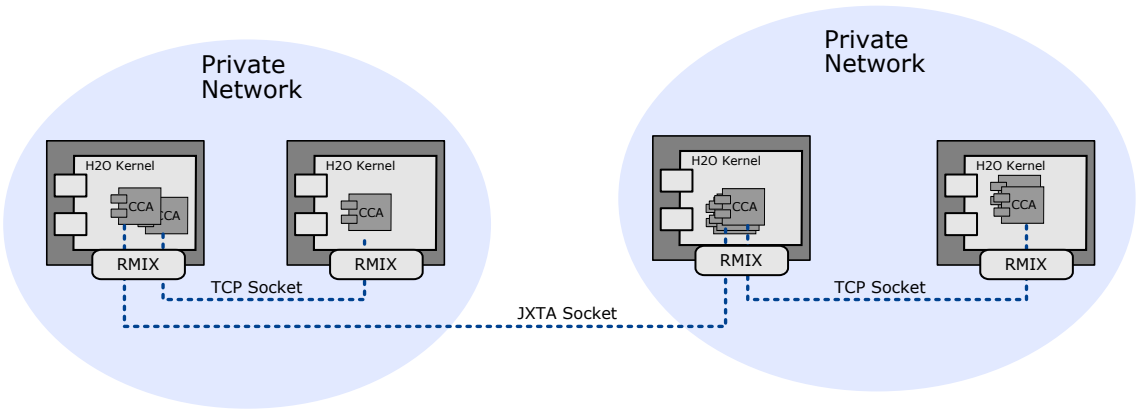


Figure 8.3: RMIX communication library in P2P environment

- Ad-hoc collaboration of shared resources - virtual computing groups (by using peer groups in P2P network);
- Compute nodes drawn from clusters and Grid nodes behind firewalls can participate in the computation.

8.3.3 Advantages of combining H2O with JXTA

Combining H2O with JXTA can lead to an interesting distributed resource sharing platform operating in a P2P manner. By combining H2O and JXTA models, resource sharing will occur between a single resource owner and a resource user. Moreover, the nature of the H2O system will allow services in containers to be developed and deployed not just by their owners, but also by independent third parties or simply by clients. Thus, a resource owner will merely decide who is allowed to deploy or use services and when. Interesting features can be achieved by applying group management in the JXTA network. By using the H2O metacomputing framework mixed with JXTA, resource owners are able to decide about their kernels' working group. As it is possible to create secured peer groups, the framework can facilitate creation of kernels groups focused on different classes of users or applications.

8.3.4 H2O in JXTA environment - design and implementation

In order to enhance H2O with the ability to operate within a P2P environment, H2O was integrated with the JXTA P2P system. The remainder of this section, highlights the most important details of RMIX-JXTA and H2O-JXTA integration.

Integration of JXTA and RMIX is achieved by implementing client socket factory and server socket factory interfaces defined by the RMIX communication library. By using the JXTA Sockets abstraction in the mentioned socket factories, the RMIX framework will communicate on top of the P2P network.

To denote the RMIX endpoint address in the P2P environment it was decided to use a flat, virtual addressing schema:

```
<string endpoint address>
  [@<group>]
```

In its simplest form, an address can be a simple string such as *rmixEndpoint*. Such addresses belong to JXTA *NetPeerGroup* and are accessible globally. Alternatively, the address can be narrowed down to a user-specified group, by providing an optional part - the definition of a private group name. This approach enables flexibility (e.g. custom security control) at the group membership level, by supporting groups that use custom implementations of Membership Service. For example, the address *myEndpoint@cGroup* denotes an RMIX endpoint with the address *myEndpoint* that is located in the *cGroup* JXTA group. When using a group with security control, a valid address of the RMIX endpoint may be the same as above, but the user should provide an AuthenticateCallback implementation that will be used by the group membership mechanism whenever there is a need to retrieve any input information (such as login or password).

From the RMIX user's point of view, the JXTA functionality is completely hidden. The user's responsibility is limited to providing the JXTA socket address (usually embedded into a serialized stub received from a naming service or from another method invocation). The RMIX recognizes the endpoint as JXTA-specific and delegates transport-related activities to JXTA socket factories. The system automatically connects to the JXTA network (becoming a JXTA peer), joins a group (if needed), manages the rendezvous status of the current JXTA peer, and connects to the JXTA socket specified via the provided address (refer to Fig. 8.3).

Once the integration of RMIX with JXTA is successfully finished, the H2O-JXTA integration is straightforward. In order to H2O in the P2P environment, there is no need of performing any additional steps. The user's responsibility is limited, as above, to providing an appropriate JXTA socket address. H2O, since its communication layer is based on RMIX, will manage JXTA connectivity transparently for users (please refer to Fig. 8.3). The only change required in the H2O configuration is the selection of the appropriate communication type.[1]

Last, but not least, integration of the above solutions with the MOCCA component environment should be mentioned. As MOCCA relies solely on H2O as a

[1]The implementation of H2O-JXTA integration was performed by Paweł Jurczyk and Maciej Golenia, co-authors of the papers [97, 98]

platform, all the underlying details of RMIX and JXTA are hidden from the component framework. MOCCA can thus use the JXTA transport in a transparent way, the only noticeable change being the usage of JXTA-based addresses in the URIs of H2O kernels and the *ComponentIds* of components. Therefore, MOCCA components can use all the available private networks, as seen in Fig. 8.2 and 8.3.

The solutions presented above were integrated with H2O and are part of the standard H2O software distribution. In the same way, it is possible to use JXTA-enabled H2O with MOCCA, which only requires a change in the H2O container configuration. For more details and performance tests, please refer to the published papers [97] and [98].

8.4 Conclusions

This chapter described the methods for integrating the component-based programming environment with production Grid infrastructures. The issues related to this problem are twofold: first, a mechanism for deployment of component containers onto the infrastructure is required, and second, a way to provide communication between components located in disparate private networks is needed.

For the first mechanism, the author propose a pool of transient H2O kernels to be deployed and managed on the production job-submission infrastructure. This allows creating a user-centric virtualization layer of component containers, upon which the actual application components can be deployed using standard framework mechanisms. The pool of resources can then be used as a basis for all higher-level programming models, opening potentially interesting avenues of research related to optimization of component placement, in the case of ADL-based composition (see Chapter 5), or scheduling, in the case of the scripting approach (Chapter 4).

To provide an appropriate communication mechanism, there is a need to adapt RMIX to run over the JXTA virtual network since RMIX is an underlying communication substrate of H2O. It was shown that this integration is possible due of the extensibility of RMIX, which allows using JXTA socket implementations. The result of this integration is a fully operational RMI implementation running over a JXTA P2P network, where methods can be invoked on remote objects located behind firewalls or NATs, which is not possible in traditional RMI systems. This, in turn, allows MOCCA components to communicate with their component peers located in private NAT networks, including those of computing nodes of clusters comprising the production Grid infrastructures.

Prototypes of the proposed solutions have been developed to demonstrate the feasibility of the approaches. Technologies such as HDNS for resource discovery and JXTA for communications proved useful for these purposes. Obviously, the proposed solutions require further development to be widely deployable and usable, but they

can serve as a proof of concept and as indications on how to bridge the emerging component- or service-based programming models with existing production Grid infrastructures oriented on processing simple batch jobs.

The new methods of creating virtual overlay P2P networks of component containers over a production Grid infrastructures described in this chapter constitutes an important element of the component-based methodology proposed in this thesis. Although the proposed approach requires more development work to improve the stability and usability of the solution, the author is convinced that it remains an interesting contribution to the research on programming and running applications on the Grid.

The approach to deployment of component applications on Grid infrastructures was published in [119], while JXTA integration is presented in [97] and [98].

Chapter 9

Evaluation: Applications and Tests

All the preceding chapters of the thesis described the methods and tools comprising the proposed component-based methodology for programming and running scientific application on the Grid. In this chapter the author reports on the tests and experiments which were conducted during the development of the resulting environment. They demonstrate both the usability of these concepts and the applicability of the entire environment for real applications.

9.1 Introduction

The goal of this chapter is to describe the applications and tests which were performed in order to demonstrate that the proposed component-based methodology *effectively* supports the scientific applications on the Grid, i.e. to verify the main research *thesis*. The chapter begins with a description of applications which were developed or adapted to the component-based environment and a discussion of the advantages of following the component approach. The remainder of this chapter describes the experiments which were conducted to measure the specific properties of the proposed solutions, such as performance and scalability.

The first application is the Application Flow Composer which was ported to MOCCA from another CCA-based framework. It serves to validate that the *base component environment* – MOCCA – works properly and that it preserves *compatibility* with other frameworks.

The next application is a *scientific* program which simulates the formation of clusters of gold atoms and which was developed from the ground up using the MOCCA framework. The advantages of applying a component-based approach are presented, together with a description of how the application was successfully *deployed* on a testbed of the European CrossGrid project and on the Grid'5000 infrastructure.

Another application, which demonstrates the usage of the proposed component environment, is a sample cellular automata computation system, parallelized using the domain-decomposition technique. This shows that the component-based

approach can also be used to model more *tightly-coupled* applications.

To demonstrate the advantages of the *high-level scripting approach* for application composition, sample experiments from the ViroLab virtual laboratory are shown. One of them is a data mining application, which uses algorithms such as classifiers, association rules generators, etc. from the Weka [187] library, wrapped as MOCCA components.

The second part of the chapter describes *benchmarks*, beginning with those used to verify the *performance* of the base component environment (MOCCA). The goal is to present this environment in comparison to the XCAT3 CCA framework and to measure the performance of both frameworks in a local and a transatlantic network setup, to demonstrate the benefits of MOCCA and of applying RMIX as a *communication* library.

To test the proposed method of *dynamic component deployment* on the pool of shared resources, the sample gold cluster application was deployed on the CrossGrid testbed. Results of experiments with deployment on the Grid'5000 infrastructure are also provided.

To demonstrate the applicability of the MOCCA framework to large-scale deployments, specific benchmark applications were deployed on the Grid'5000 testbed. Their goal is to demonstrate that the MOCCA framework behaves correctly and does not introduce significant performance overhead when hundreds of components are running and communicating on the Grid infrastructure.

The gold cluster simulation application also serves as an example of *interoperability* between GCM and CCA component models. It shows how an external component, developed in ProActive, can be plugged into the running MOCCA application, and how this connection can be established using the *scripting* approach.

Finally, sample tests were performed using the GridSpace application optimizer. The performance tests were performed to validate the need of an optimizer and to show how various optimization algorithms can influence the overall application performance.

The tests and experiments were prepared in such a way as to verify the selected properties and features of the proposed methodology and the supporting environment. A wide range of component configurations and possible usage scenarios were selected to demonstrate the universal applicability of the proposed methodology and the benefits of using the proof-of-concept implementation.

9.2 Application Flow Composer example

As a first application of the MOCCA framework (see Chapter 6), the author decided to use an Automatic Flow Composer (AFC) [26], taking into account our previous experience with this application. AFC is a system that was initially designed and

developed for the XCAT framework. Its goal was to automatically compose a description of an application workflow. This description is an XML document which contains the definitions of components, their ports and connections between them. This document may be supplied to a tool (the workflow engine) that automatically creates component instances, sets up appropriate connections and executes the resulting application within the framework. The AFC is able to automatically generate complete workflow description(s) given some initial and final constraints on the components involved plus access to a registry containing the list of available component descriptions. AFC and AFC2 [27] were the first prototypes on the path to exploiting semantic descriptions of components for more accurate matchmaking, and supporting workflows built from Web services, which resulted in the WCT [80] system, developed within the K-WfGrid project.

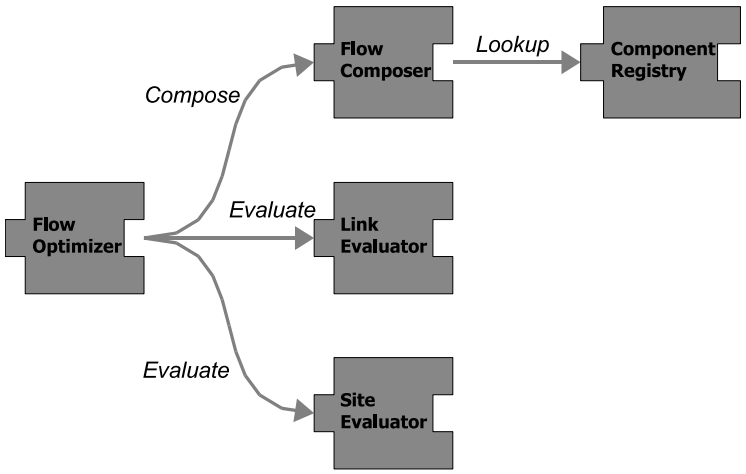


Figure 9.1: Flow composition example

Following this short introduction to AFC, we will now describe experience with porting and executing it on MOCCA. The AFC consists of five components connected together, as seen in Fig. 9.1. The initial component, called Optimizer, selects the best workflow descriptions from those generated by the Composer component. The Optimizer uses two additional monitoring components, one for evaluating the performance of remote sites where components may be instantiated, and another for estimating the quality of their Internet connections. In this test, performance data was not relevant, hence the evaluator components produced their answers basing on artificial pre-generated data.

The conclusion is that migration of AFC from XCAT to MOCCA was an easy task. This is because both frameworks are CCA-compliant, with only minor departures which required slight code modifications (e.g. removing dependencies on XCAT Java packages and framework-specific properties). Setting up an application

on remote sites in the case of XCAT requires either ssh interactive access (with public key authentication) or the Globus GRAM. In the case of MOCCA, it suffices to have access to the H2O kernel. The deployment of component code is then fully automated – the user only needs to specify the location of the JAR component file. AFC performance was similar in both frameworks, due to the fact that the test application was not communicationally-intensive, and also the data exchanged between components consisted of XML documents.

9.3 Gold cluster formation

Clusters of atoms are an interesting form in between isolated atoms or molecules and solid state. Modeling of clusters involves several computationally-intensive energy minimization methods such as Molecular Dynamics Simulated Annealing (MDSA) or the Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) method, as well as choosing an empirical potential [186]. An optimal result depends on the number of possible iterations and initial configurations for each simulation run. Moreover, the process of energy minimization can also be subject to optimization by tuning the parameters of the simulation, which can be achieved by gathering statistics from many execution runs. This leads to an interesting model calibration scenario, which is a good example of highly computationally-intensive scientific applications.

The original application (sequential code written in C) has been rewritten in Java and its functional modules have been divided into separate components. Early results indicate acceptable scalability of this application when tested on a single cluster [29].

The component-based application for simulating formation of gold clusters has evolved over time. Below, two important versions of the application are described, the first one for simple minimization and the second one involving tuning of application parameters.

For the first set of experiments, the component configuration is shown in Fig. 9.2. The *Starter* component is responsible for coordinating the work of other components. *Configuration Generator* creates the initial random configurations of atoms, which are then consumed by multiple *Simulated Annealing* components, performing the actual minimization process. The *Configuration Generator* and *Simulated Annealing* components may be used for both sequential and distributed configurations, since they do not have multiple ports. The *Storeroom* component is responsible for storing all achieved configurations and may be used to derive results statistics. A single *Molecule* port is devoted to exchanging data between components. The *Storeroom* component was initially designed to support a single *Molecule* provider, and to keep it simple, the decision was made to add a separate *Gather* component which handles multiple connected components and passes their results to the *Storeroom*.

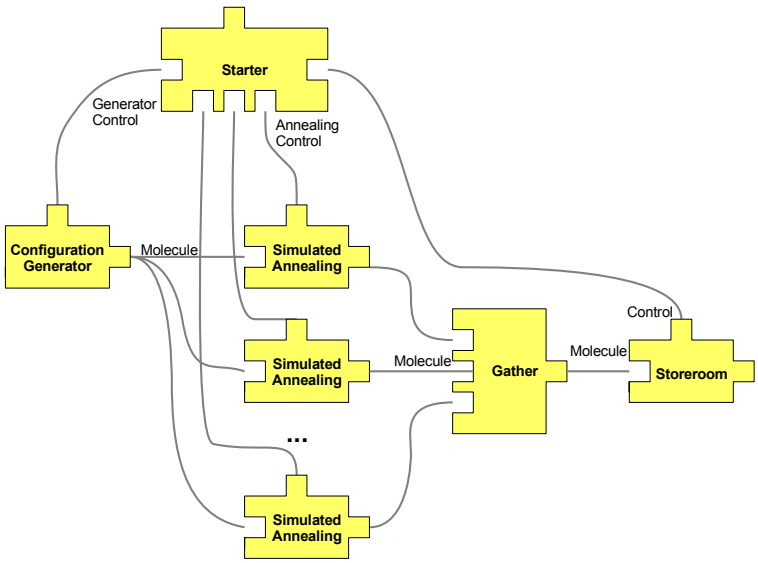


Figure 9.2: Configuration of components in the gold cluster application (first version).

Further rationale for keeping *gather* functionality in a separate component instead of incorporating it in the *Storeroom* is the possibility of building a hierarchical tree of gather components, which may be required when deploying the application on a large number of nodes.

The second version, shown in Fig. 9.3, is more elaborate. The *Simulated Annealing* components were extended to use the externally provided *Annealing Function* which represents the strategy of cooling the system and influences the optimization process. Such a function can be provided by a specialized *Annealing Function Manager* component, which gathers statistics about the optimization process from the *Simulated Annealing* components in order to improve the cooling function. Additionally, the *Local Minimization* component is connected to the *Storeroom* to improve the results using the L-BFGS method (using the JAT [133] library developed at NASA). For interactive visualization, a prototype version of the *Output Generator* component was also developed, using the Jmol [85] visualization library (not shown in the diagram). Another change to the first version was replacing control ports with CCA *parameter ports* which allows reading or writing custom properties in a standard way.

It can be observed that the *Molecule* and *Statistics* ports, together with their corresponding *Gather* components, have similar functionality. To facilitate development, a common abstract port class called *buffered port* was introduced, which helps

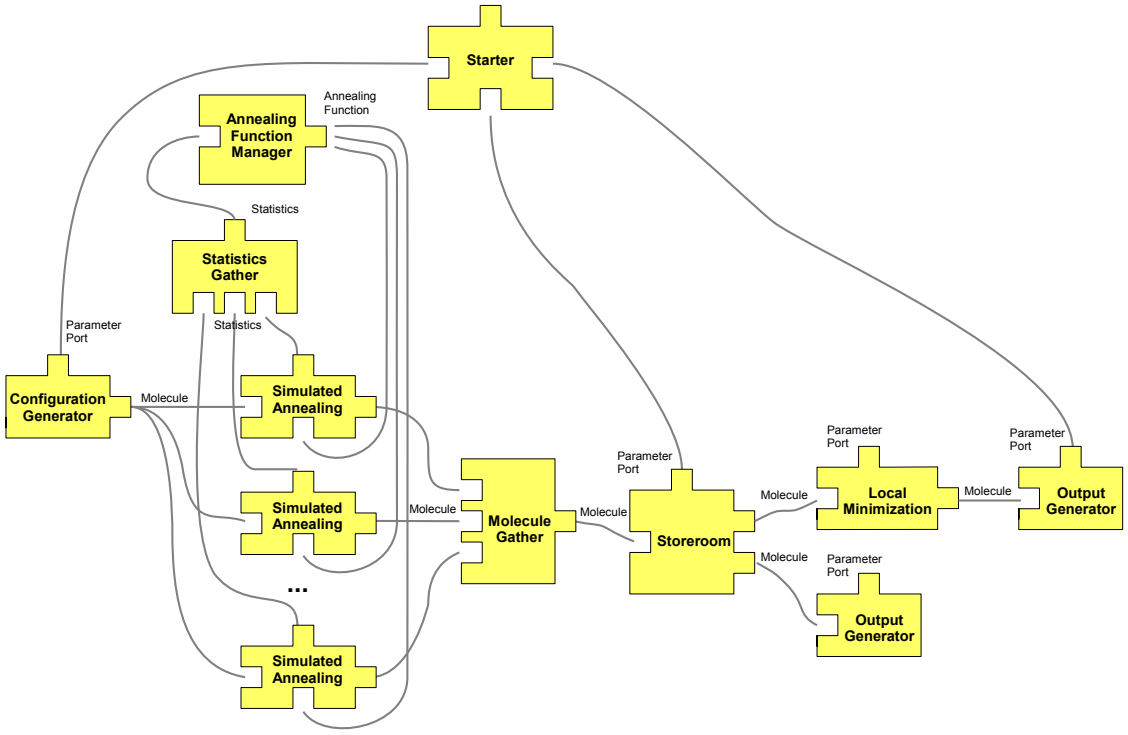


Figure 9.3: Configuration of application which enables tuning its parameters (version 2)

manage the queue of data items to be processed. The gather functionality has been also abstracted so that it can be reused in other applications.

The benefits of decomposition of the application into components are twofold. Firstly, it allows for more flexibility of application assembly when experimenting with different minimization methods and also with different configurations and deployments. Secondly, the performance penalty incurred by using Java may be outweighed by its portability, leading to the possibility of using many distributed resources for computation. When deploying the application on a *blade* cluster at AGH (16 nodes with dual Intel Xeon 3 GHz processors) it was possible to reproduce all the results from the original paper [186].[1]

[1]The physics-related aspects of the application and the implementation of specific components was performed under the author's supervision by Michał Placek, co-author of the papers [29, 119].

9.4 Weka experiments in ViroLab

The ViroLab virtual laboratory [28, 185] is a system for collaborative construction and execution of experiments in computational science. It is focused on, but not limited to, infectious diseases caused by such viruses as HIV. The usage of virtual laboratory includes such experiments as analysis of HIV genomic structure and prediction of virus resistance to various types of drugs [160, 28].

MOCCA is one of the supported middleware technologies, and the GridSpace scripting engine, as described in Chapter 4, is used as a core system for application execution. The system was applied to constructing and executing real-life examples. Below, we show how the components can be used to perform a data mining experiment using the Weka [187] library wrapped in components. Two versions of Weka components are described: a simple one and an advanced version using URLs to pass datasets.

```
require 'GridOperationInvoker/Core/g_obj'
dataProvider =  GObj.create('gridspace.weka.WekaGem')
A = dataProvider.loadDataFromDatabase(DATABASE, QUERY, USER, PASSWORD)
B = dataProvider.splitData(A, 20)
trainA = B.predictingData
testA = B.testingData
classifier = GObj.create('gridspace.weka.OneRuleClassifier')
classifier.train(trainA, ATTRIBUTENAME)
prediction = classifier.classify(testA)
predictionQuality = dataProvider.compare(testA, prediction, attributeName)
```

Figure 9.4: Sample data mining application script

Fig. 9.4 shows an experiment which uses simple Weka components. The experiment uses two Grid Objects, *dataProvider* and *classifier*. The former provides a remote interface to retrieve integrated data from various databases in the ARFF format [187], allows user to split data into two parts and evaluates the similarity of two data sets. The latter is a data classifier: once trained with sample data, it predicts one attribute in the given data set using the One-Rule algorithm. The classifier uses the Weka data mining library. In the script, data is first retrieved and then split into training and testing sets by the *dataProvider*. Subsequently, one set is used to train the *classifier* which is then used to classify the other data set. Finally, *dataProvider* is used again to estimate the quality of classification. In this experiment, the *dataProvider* Grid Object is implemented as a Web service, while the *classifier* is a stateful MOCCA component, dynamically deployed on the computing resource.

The experiment described above, albeit simple, demonstrates several benefits of the component-based approach. First, the *Classifier* component is a *stateful* entity, which is *created (deployed) on demand* and can use the available resources (H2O

kernels). An instance of the classifier is created for each experiment run. It can also be used in *collaborative* scenarios, when a classifier is trained by one experiment (user) and then published for use by other experiments (users).

The second version of Weka components allows for more flexible creation of various experiments. Key functionality elements of Weka, such as classifiers, association rules, clustering algorithms and filters, were wrapped as components. To provide better performance in terms of transferring and storing datasets, it was decided to use the HTTP protocol and a WebDAV server. The components can now retrieve the datasets from any remote URL and store the results on a WebDAV server, which makes them, again, available via URL. Such *pass-by-reference* approach is very convenient, since the whole (potentially large) dataset does not have to be directly passed through the GridSpace engine.

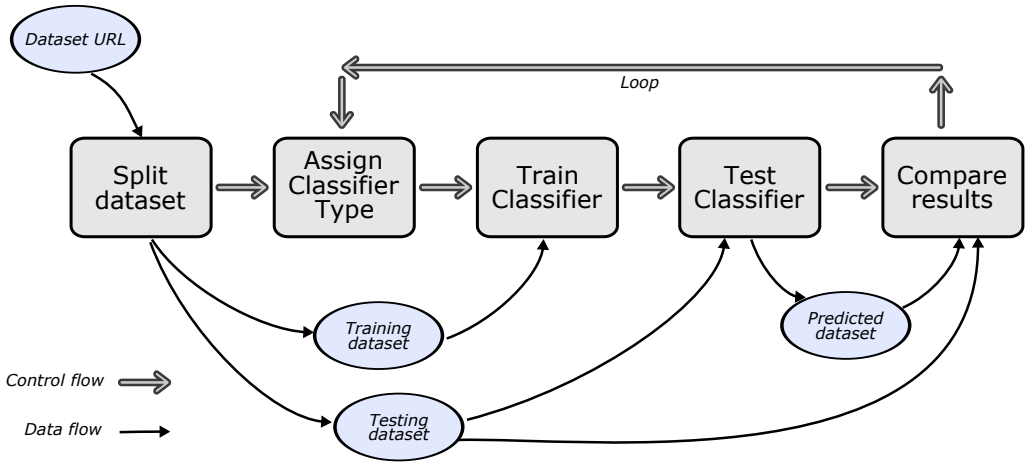


Figure 9.5: The data and control flow for the sample script demonstrating the use of the Weka Data Mining application which uses MOCCA components.

Fig. 9.5 presents the scenario of an experiment which can be used to compare the performance of several classifiers from Weka on a sample dataset. It is implemented as a script shown in Fig. 9.6. The script demonstrates how to create an instance of a classifier component, supply it with a specific algorithm and perform the classification, measuring the time and accuracy of the predictions. The scripting approach allows easy creation of complex experiments using constructs such as loops, thus providing effective and flexible experiment *steering*.[2]

[2]The author acknowledges the support of the ViroLab team at Cyfronet including Tomasz Bartyński, Marek Kasztelnik and Tomasz Jadczyk who helped implement and integrate the Weka components and experiments with the ViroLab virtual laboratory.

```
  Classifiers = [ 'weka.classifiers.rules.ZeroR',
        'weka.classifiers.rules.OneR',
        'weka.classifiers.functions.SimpleLogistic',
        'weka.classifiers.trees.Id3',
        'weka.classifiers.functions.LeastMedSq',
        'weka.classifiers.rules.Prism',
        'weka.classifiers.functions.Logistic',
        'weka.classifiers.lazy.IBk',
        'weka.classifiers.trees.J48',
        'weka.classifiers.lazy.KStar']

  puts 'Compare classifiers'

  wekaURLgem = GObj.create('cyfronet.gridspace.gem.weka.WekaURLGem')

  classifier = GObj.create('cyfronet.gridspace.gem.weka.WekaClassifier')

  dataURL = 'primary-tumor' #address in WebDav repository contains data
  splitDataName = 'split-primary-tumor'
  splitURLData = wekaURLgem.splitURLData(dataURL, '', splitDataName, '', 50)

i = 0
10.times do
  classifier.assignClassifier(Classifiers[i], java.lang.String[1].new)
  learning_time = classifier.trainURLdata(splitURLData.trainingURLdata, '', 'class')
  puts 'Learning took: ' + learning_time.to_s + '[ms]'

  classifiedData = classifier.classifyURLdata(splitURLData.testingURLdata, '', '', '')

  classificationPercetnage =
      wekaURLgem.compareURLData(splitURLData.testingURLdata,'', classifiedData,'', 'class')

  result = classificationPercetnage.to_f * 100.to_f
  puts Classifiers[i] + ' was correct in: ' + result.to_s + ' percents'
  puts ''

  wekaURLgem.deleteURLdata(classifiedData)

  i = i + 1
end

  wekaURLgem.deleteURLdata(splitURLData.trainingURLdata)
  wekaURLgem.deleteURLdata(splitURLData.testingURLdata)
```

Figure 9.6: Advanced data mining application script

9.5 Domain decomposition example

The MOCCA component framework was also used to develop a tightly-coupled model computing application, which simulated cellular automata (John Conway's Game of Life) following the classic domain decomposition technique. The goal was to separate the communication part of the application, which is generic, from the application-specific computing part. This was done using the approach proposed in Section 5.2.4, where communication components, connected together in a 2-D grid, are used to perform the synchronization and exchange of boundary data between

computing components (see Fig. 9.7). The communication components are reusable for any application, and the application developer has to provide the implementation of the computing components which provide the defined *ComputationPort*. Since the corresponding pairs of components can be deployed in the same kernels, their connection is local and does not introduce much communication overhead.[3]

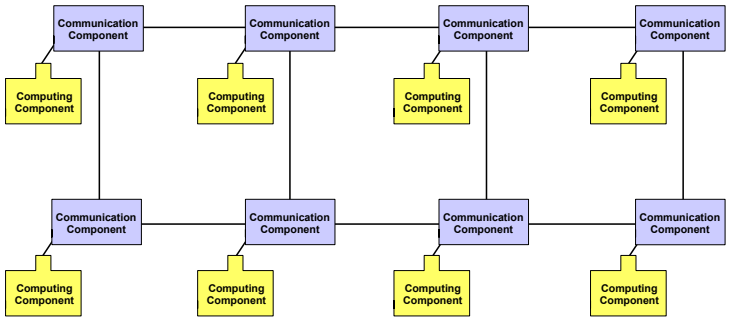


Figure 9.7: Domain decomposition – configuration for 2x4 computing components

The results of experiments with the application deployed on a *blade* cluster at AGH (16 nodes with dual Intel Xeon 3 GHz processors) are shown in Fig. 9.8. The goal was to measure the relative *scaled* speedup, i.e. speedup obtained when the problem size grows with the number of processors while the data set size per one processor remains constant. The experiments were performed for several data array sizes. We can observe good behavior of this application – thus the speedup increases together with the size of data array and the best parallel efficiency is reached for the largest data set size (8000×8000 bytes) for 8 computing nodes.

Results from the domain-decomposition benchmark demonstrate that the component-based methodology can be effectively applied to such tightly-coupled problems. The component approach allows us to create reusable components which constitute the *skeleton* of the application, while it is possible to plug in various domain-specific implementations of computing components. The results of runs on a single cluster are also promising and suggest that component-based solutions do not have to introduce significant overhead. More detailed measurements and larger deployments are left for future work.

[3]The application components used in the tests described here were implemented by Przemysław Pelczar under supervision of the author.

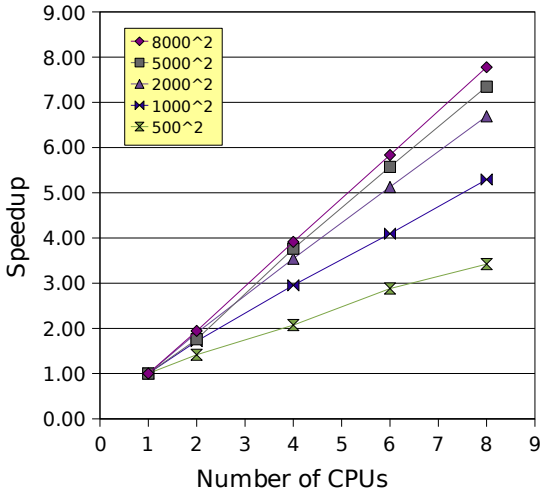
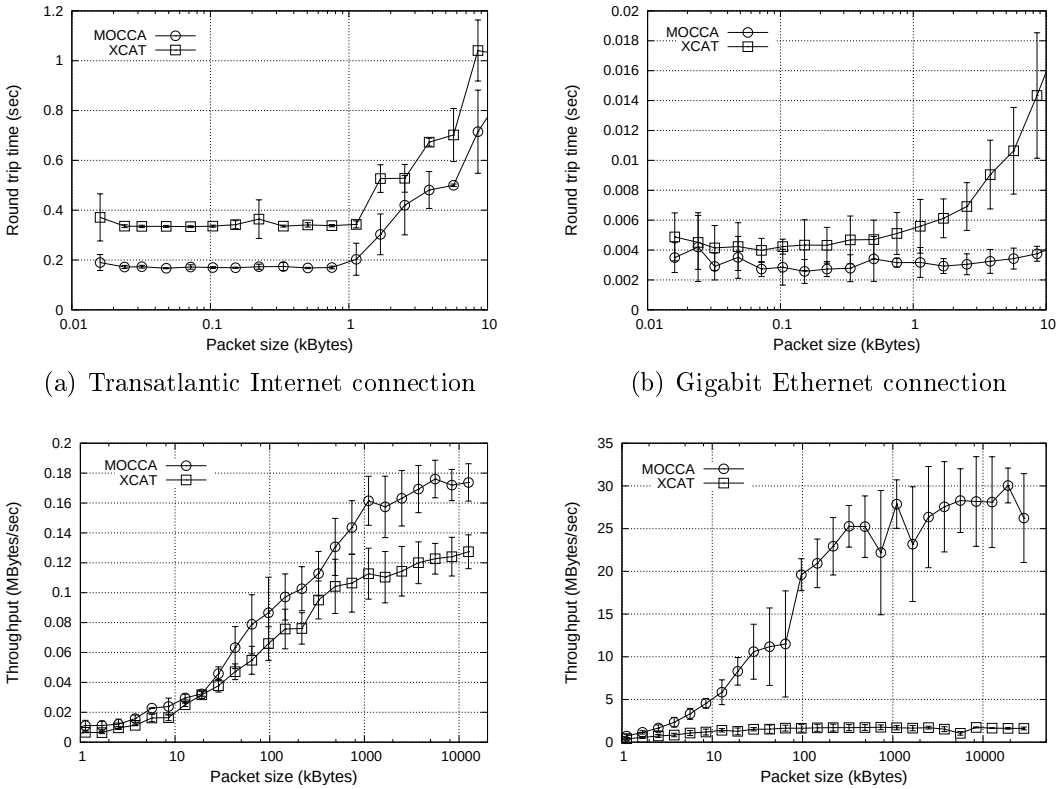


Figure 9.8: The results of running domain decomposition on a Blade cluster. The graph shows the relative speedup with respect to the number of processors. Each data series shows results obtained for different size of the data array (a square array of bytes).

9.6 Communication-intensive benchmark

To measure the performance of the invocations of methods on CCA ports in MOCCA, a test was conducted on a highly simplified communicationally-intensive application, modelling real scientific computation. The application consisted of two components: a model computation engine (such as FFT), performing calculations on an array of `double` numbers and returning the modified array as a result, and a client component utilizing the engine. Two configurations of components were tested for differing network connections. The first testbed consisted of two machines, a workstation in Atlanta, USA, and a cluster node located in Krakow, Poland, so that communication utilized regular transatlantic Internet links. The second testbed was built of two 2.4GHz Pentium4 PCs connected by a Gigabit Ethernet LAN. On both testbeds, the roundtrip time for different packet sizes was measured.

Fig. 9.9 shows the results obtained in both networks by MOCCA and XCAT frameworks. At the top of the figure, results for small packet sizes are presented in the form of roundtrip time. In this experiment, the network latency turns out to be a major bottleneck. Nevertheless, XCAT introduces a significant additional overhead, attributable to the size of SOAP headers, outweighing the payload, as well as time needed to open the connection. In MOCCA, this overhead was much smaller, owing to more efficient protocols and TCP connection pools used by RMIX. The two



(a) Transatlantic Internet connection

(b) Gigabit Ethernet connection

(c) Transatlantic Internet connection

(d) Gigabit Ethernet connection

Figure 9.9: Round-trip time and throughput measured for invocations between components.

bottom plots illustrate throughput achieved for large data packets. On the wide area connection, where network bandwidth is a bottleneck, MOCCA performs about 30% faster than XCAT due to its more compact data encoding (binary versus Base64). In the case of the Gigabit LAN, CPU saturation was observed, indicating that the problem became computation-bound. MOCCA was able to achieve 30MBytes/sec (approximately 25% of the theoretically-available bandwidth), which is considered a promising result (please note that measured time covered the entire two-way RMI call requiring copying data arrays into communication buffers on both sides). This suggests there is room for possible improvements, by eliminating unnecessary buffer copying. XCAT performed very poorly, utilizing only about 1.7% of the available bandwidth. The plots also show the standard deviation, computed on the basis of 10 runs of the same test. For the transatlantic network, the uncertainty was caused by the multiplicity of users sharing the connection; the results, however, were reproducible at different times of day. In the case of the Gigabit Ethernet, significant

variations of throughput were observed for certain message sizes. Since the variation patterns depended on the garbage collection settings, it was possible to attribute these artifacts to the asynchronous Java GC, interfering with the measurements. The results shown in the figure were obtained with SUN J2SDK 1.4 running in server mode and with incremental garbage collection turned on. All tests were conducted under the Linux operating system.

Having understood most of the factors leading to the overhead and variance, the conclusion from the tests is that the performance of RMIX-based MOCCA seems promising not only for widely-distributed, but also for more tightly-coupled applications.

9.7 Application deployment experiments on CrossGrid and Grid'5000

This section describes the experiments with the method of dynamic deployment of component containers onto a heterogeneous infrastructure, as described in Chapter 8. As a test application, the *gold cluster formation* simulation described in Section 9.3 was used. The testbed consisted of two standalone machines at CYFRONET, Krakow, accessible directly via SSH, a cluster node at CYFRONET, accessible using PBS, and the European CrossGrid infrastructure running LCG middleware with a Resource Broker located at LIP in Lisbon, Portugal. When submitting jobs via the Resource Broker there was a need to restrict the list of resources to only those Grid sites which allowed jobs (in our case – H2O kernels) to open TCP ports for incoming connections from the outside world. Therefore it was possible to use cluster nodes at PSNC, Poznan and IFCA, Santander, Spain. The machines at CYFRONET were Intel Xeon CPU 2.40GHz computers, with 512 MB RAM running Red hat Linux 7.3, and similar configurations were available on remote Grid sites. Sun J2SDK 1.4.1 was available on all machines, therefore no additional JVM installation was required.

To generate a pool of H2O kernels, an HDNS server was set up on one of the machines at CYFRONET, and another machine hosted the kernel on which the *Starter* component had to be deployed in order to have direct access to results and timing statistics for the application run. Subsequently, it was possible to spawn more H2O kernels on local cluster nodes and on Grid sites using scripts, as described in Chapter 8. By using the pool of H2O kernels, it was possible to deploy the distributed component application, creating one computationally-intensive *Simulated Annealing* component for each available kernel. It should be noted that once the pool of kernels was created, deploying the application was as simple as on a single cluster, requiring only minor changes to the deployment script. These changes involved replacing a

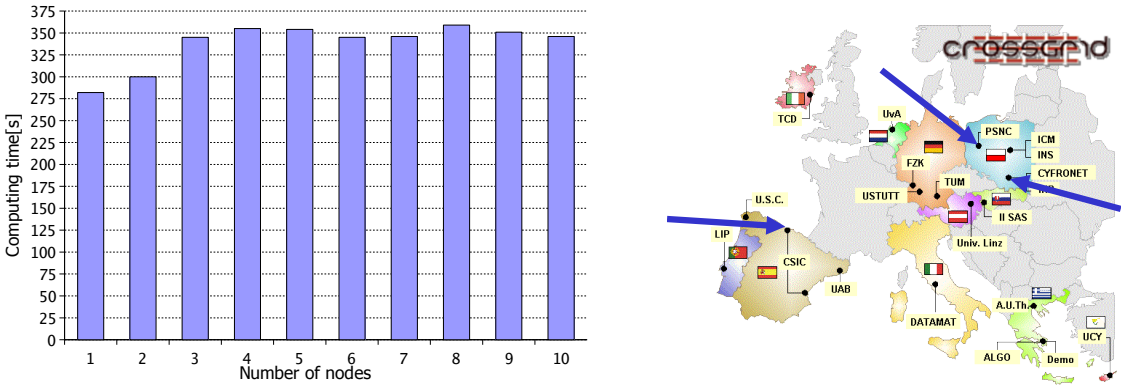


Figure 9.10: Results achieved on a sample pool of heterogeneous resources, where the problem size grows with the number of computing nodes.

static list of machines with an appropriate invocation of the scheduler.

Fig. 9.10 shows the sample application's run time depending on the number of computing nodes. On each of the nodes there was one *Simulated Annealing* component deployed. The total number of initial configurations for the simulation was equal to the number of computing nodes, so each of the components was computing one full simulation run for a single molecule. By increasing the number of resources in the pool the user was able to simulate more molecules, which is important from the application's point of view (in order to gather better statistics). It can be observed that the total computing time does not increase substantially with the number of nodes, which approximates an ideal case where the time should remain constant. It should be noted, however, that the goal of the tests was to show proof of concept for our approach to building a user-centric pool of resources for distributed component applications, and not to run systematic performance measurements on the Grid, where speedup is not as clear as on a parallel machine or on a single cluster [88]. However, the results achieved on this sample pool of resources are quite promising for such an ad-hoc testbed and base upon the nature of the tested application (which was not communication-intensive).

By following a similar approach, it was possible to deploy the simulated annealing application on the French Grid'5000 testbed. The application used three clusters located at Sophia-Antipolis, Bordeaux and Orsay. The application was successfully deployed on up to 220 computing nodes (cores) and the computing times for the molecules of 20 atoms were measured. Table 9.1 presents only sample computing times for selected numbers of molecules which were processed. The number

cores	79	98	98	98	98	218	220
molecules	62	20	40	20	58	78	56
time, seconds	135	125	173	112	117	271	169

Table 9.1: Execution times for sample runs on Grid'5000. The first row shows the number of computing nodes (cores) which is equal to the number of deployed components. The second row shows the number of molecules successfully computed. The third row shows the total computing time in seconds.

of computed molecules is smaller than the number of deployed components, which means that only a part of the components performed actual computing, while others remained idle. It can be shown that the computing time for all the experiments remained on the same order of magnitude and temporal variations can be attributed to the heterogeneity of the testbed, including different processors and JVM versions which were available. Unfortunately, more systematic measurements were impossible due to instabilities found in the application and heavy load of other jobs running on the testbed.

The conclusions from experiments on both CrossGrid and Grid'5000 testbeds are that the proposed method of virtualizing heterogeneous resources by instantiating a pool of component containers on them can provide an effective method of aggregating the power of these resources. Once their heterogeneity is hidden, they can be used in a standard way, as a basis for deployment of application components.

9.8 Scalability experiments on Grid'5000

The purpose of the following experiments, which were run on the French Grid'5000, testbed was to test and analyze the scalability of the MOCCA environment on a large number of nodes. A benchmark application was constructed to allow extracting important system metrics, such as time of deployment, connection, invocations on collections of ports and cleanup of components. The characteristics of the clusters used in the experiments are presented in Table 9.2.

The structure of the application is shown in Fig 9.11. The *Starter* component is connected to the collection of *Forwarder* components, which in turn are connected to a single *Echo* component. The *echo()* operation on the port of connected components consisted of passing and returning a string message, several bytes of length. The components were created using the *MultiBuilder* mechanism, as described in Section6.4. In ***version 1*** of the benchmark, the *Starter* component was executing the echo operation *sequentially* on all ports connected to it.

As a first experiment, the application was run on a pool of 114 H2O kernels running on 114 nodes of 6 clusters, totalling 258 cores. The number of *Forwarder*

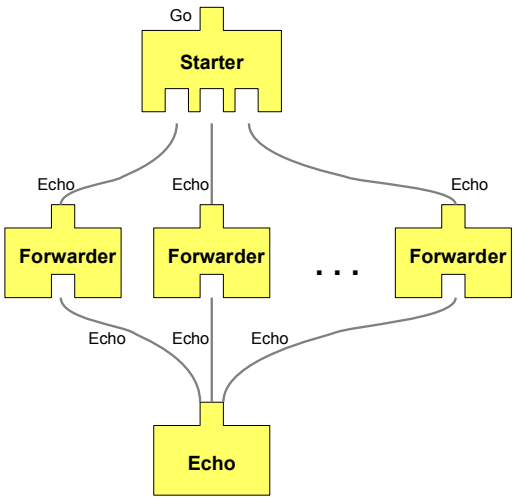


Figure 9.11: Configuration of components in the benchmark application. The number of *Forwarder* components in the collection is parametrized.

components in the collection was equal to the number of cores. The total run time (from client startup till the end of cleanup) versus the number of cores is shown in Fig 9.12. It can be seen that the growth of computing time is linear with respect to the number of components (cores). This can be explained by the fact that all operations (deployment, connection, invocation and destroying) were invoked *sequentially*. The average processing time per component was 2 seconds, which is comparable to the time of running the above application on a single node. The conclusion from this experiment is that creation of a large number of connections between components using the *MultiBuilder* mechanism does not introduce additional overhead.

machine	site	nodes	cpus	cores	CPU Type
grelon	nancy	120	240	480	Intel Xeon 5110
grilon	nancy	47	94	94	AMD Opteron 246
gdx	orsay	186	372	372	AMD Opteron 246
netgdx	orsay	30	60	60	AMD Opteron 246
parasol	rennes	64	128	128	AMD Opteron 248
paravent	rennes	99	198	198	AMD Opteron 246
paraquad	rennes	64	128	256	Intel Xeon 5148 LV
paramount	rennes	33	66	132	Intel Xeon 5148 LV
bordereau	bordeaux	93	186	382	AMD Opteron 2218
bordemer	bordeaux	48	96	96	AMD Opteron 248
chinqchint	lille	46	92	368	Intel xeon
azur	sophia	72	144	144	AMD Opteron 246
sol	sophia	50	100	200	AMD Opteron 2218

Table 9.2: Clusters of Grid'5000 which were used in the experiments.

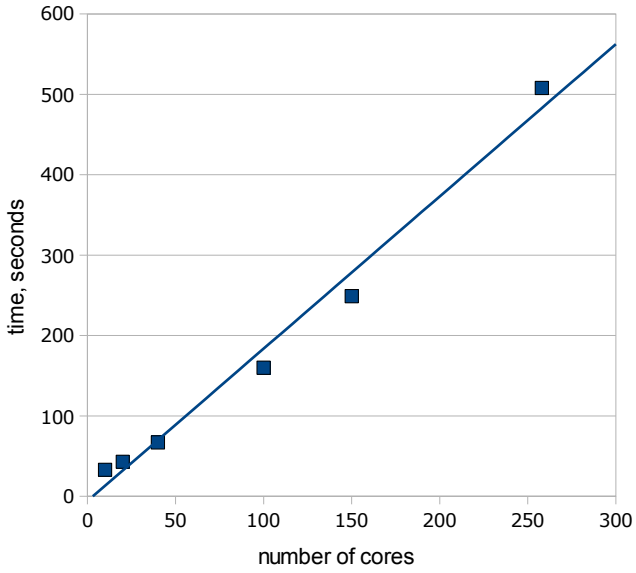


Figure 9.12: Total execution time of test application (version 1) on 250 cores, 6 clusters (gdx, bordemer, parasol, paravent, paraquad, paramount)

This means that the environment preserves scalability when handling collections of components sequentially.

The goal of the second experiment was to measure the computing time for each stage of the application. The results of the two sample runs are shown in Tab. 9.8. As can be seen, the most time-consuming stages are the creation of components and the actual computing, which is the time of passing the echo message from *Starter* through *Forwarders* to *Echo* and back again. The creation time is reasonably long, since it involves opening new sessions to H2O kernels and instantiating a new component, including classloading. The reason behind the lengthy computation time stems from the implementation of CCA `connect()` and `getPort()` methods in MOCCA. When components are connected, the *uses* side only receives a reference to the *provides* side. The actual opening of a session to the H2O kernel of the provider is performed when the user component requests a reference to the *uses* port from the framework, which is done during application execution (compute time). In the case of the benchmark application, there are two such operations per each *Forwarder*, which explains the delay and overall time.

To improve the computing time of the application, the implementation of the *Starter* component was modified in such a way that invocations on the collection of uses ports are performed concurrently and asynchronously by using the *cached thread*

n	creation	connection	computing	destroy	total
260	207	25	219	99	551
240	90	20	171	103	384

Table 9.3: Detailed measurements of application stages (version 1) for sample runs. Number of computing nodes (cores) is denoted by n and the time is given in seconds.

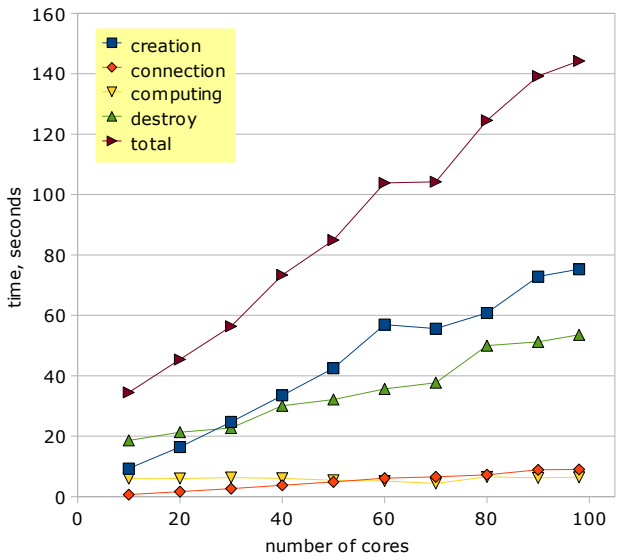
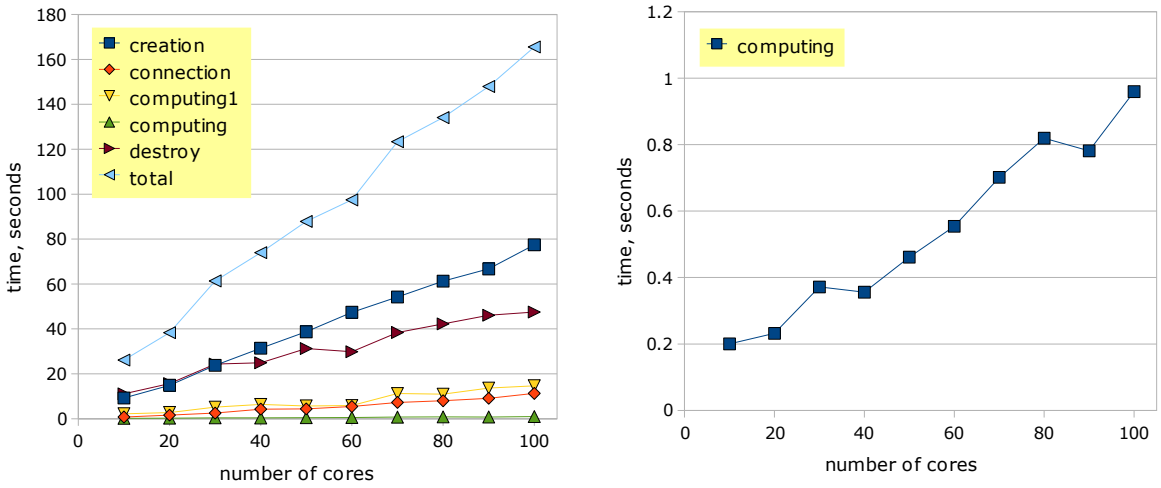


Figure 9.13: Detailed execution times of the test application (version 2) on 100 cores of 6 clusters (chinqchint, gdx, netgdx, bordemer, borderau, paravent)

pool executor mechanism from the `java.util.concurrent` package. The opening of sessions and execution of forwarders can then proceed in parallel. The results of detailed measurements of ***version 2*** of the benchmark performed on 100 cores distributed over 6 clusters are shown in Fig. 9.13. This time, the computation time is reduced to approximately 5% of total run time, while for the sequential version it was nearly 50%. The conclusion from this test is that asynchronous execution can considerably improve application performance.

In order to distinguish the opening of the H2O session from the actual remote method invocation on component ports, the *Starter* component was further modified to invoke the echo operation several times after obtaining a reference to the port (***version 3***). The time of the first invocation (labelled computing1) was measured separately from the average time of the 10 subsequent invocations (labelled



(a) Execution times of the stages of the test application

(b) Average execution time of the computing stage (enlarged)

Figure 9.14: Detailed measurements of the test application (version 3) on 100 cores of 4 clusters (grillon, gdx, netgdx, borderau)

computing). The results are presented in Fig. 9.14(a) with the computing time (enlarged scale) shown in Fig. 9.14(b). It can be seen that the computation time for 10 components (cores) is 0.2 seconds and for 100 it grows to nearly 1 s. The average network latency between clusters measured using the `ping` command was 0.017 s and the measured invocation time involves 4 such network hops. By comparing these values it can be seen that the component framework does not introduce significant overhead. It was also observed that the invocation (computation) time grows linearly with the number of nodes, which must be caused by the combined effects of the sequential nature of initiating asynchronous invocations, the single network connection from *Starter* and to *Echo*, as well as a single 2-CPU node these two components were deployed on. The invocation time can be potentially further optimized by using an efficient broadcast algorithm, which was, however, not the goal of this work.

In addition to the above described benchmarks it was possible to deploy and run the test application on 600 and 800 cores of 8 clusters respectively. The results shown in Tab. 9.4 are in agreement with the linear relation observed in previous tests, although more systematic experiments would be required to confirm this behavior for large-scale deployments on more than 1000 processor cores. It should be noted that such experiments were not easy to conduct due to the varying load on the whole Grid'5000 testbed and limited opportunities to reserve more than 1000 cores.

Experience indicates that out of the reserved nodes, ca. 10% may have some configuration problems (with the Java Runtime Environment, with network connection or otherwise), preventing the H2O kernel from starting.

n	creation	connection	computing	destroy	total
800	415	80	66	287	849
600	222	49	46	202	518

Table 9.4: The duration of subsequent stages of application deployment on up to 800 cores of 8 clusters (grelon, azur, helios, sol, gdx, netgdx, bordemer, borderau). The number of cores is denoted as n and the execution time is given in seconds.

In general, the results of the large-scale deployment experiments can be considered very promising. First, it was possible to successfully deploy, execute and clean up the benchmark application on up to 800 processor cores of 8 clusters of the Grid'5000 testbed. Comparing the sequential invocation to the parallel one results in the conclusion that it is possible to achieve increasingly better efficiency with the latter mode. The times of various steps of the application lifecycle were measured and the observed behavior was explained. Finally, the author can conclude that the component-based approach does not introduce significant overhead and the environment retains scalability even for large-scale deployments. These results are consistent with those yielded by tests of other Java-based frameworks such as ProActive [147] or Satin [183].

9.9 Interoperability and the high-level scripting composition

This section describes experiments with the interoperability solution between GCM and CCA (Chapter 7). The prototype was verified on a number of testing components using sample scripts demonstrating the functionality of the whole runtime system. It was possible to successfully integrate MOCCA components with those running in the ProActive framework.

One of the scenarios include wrapping a scientific application used for simulation of clustered gold atoms as described in Section 9.3. The test started by deploying and running the application in the MOCCA framework. Subsequently, by wrapping the `Molecule` port using the appropriate *glue* ports, it was possible to plug an alternative output generator component into the running application (Fig. 9.15). This component was developed using ProActive. Connecting the glue ports and the external component called for using the scripting approach. A sample script which creates a ProActive component and connects it to the running MOCCA component

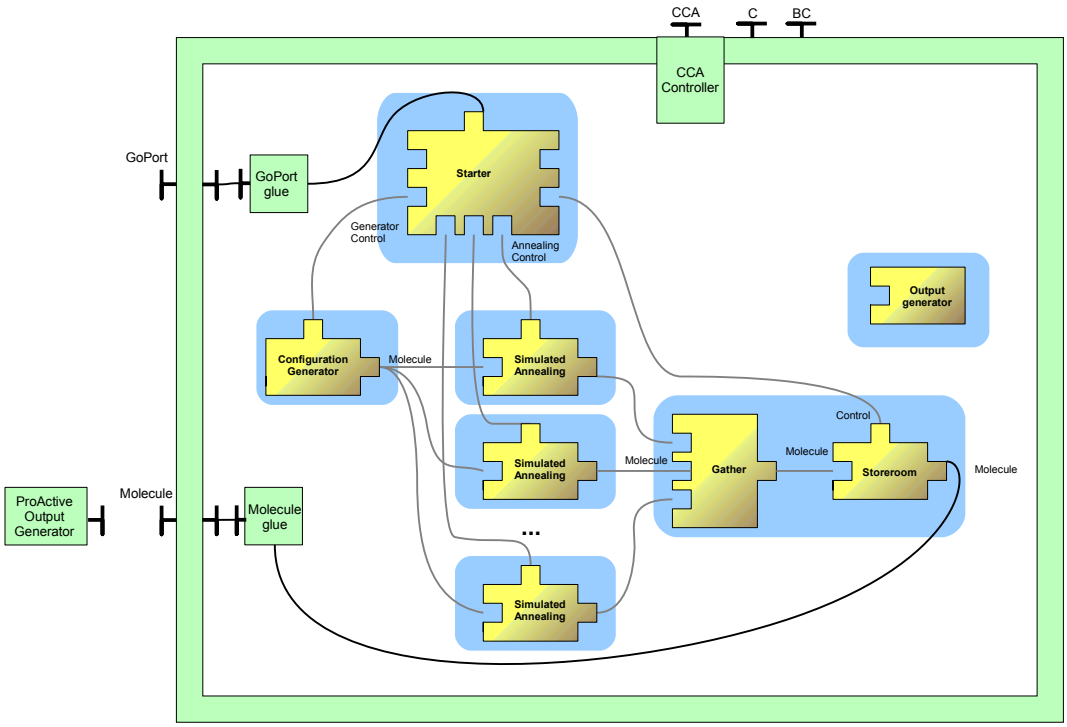


Figure 9.15: CCA simulation running in MOCCA connected to a ProActive component

subsystem is shown in Fig. 9.16 (techInfo details are omitted). In the script, it can be seen that it is possible to connect the required components and to invoke operations on them. The latter feature is used to start the lifecycle of the ProActive component and to invoke the `go()` method on the wrapped `Go` port.

This experiment demonstrates the combined interoperability on two levels: first, inter-framework interoperability and second, the advantages of the high-level scripting approach which can be used to combine CCA and GCM components in a transparent way.

9.10 GScript optimizer tests

This section reports on the tests of the GridSpace optimizer module which was described in Chapter 4. The purpose of the test was to show that the scripting approach can be subject to optimization, as well as to present its dependency on

```
#  create component from tech info

wComp = GS.createConcrete(wrapperTechInfo)
oComp = GS.createConcrete(outputTechInfo)

# bind output to the wrapper
oComp.UsesMoleculePort.connect(wComponent.MyMoleculePort)

# start wrapper component
wComp.startFc()

# start output generator component
oComp.startFc()

# Invoke Go port
wComp.go();
```

Figure 9.16: Script for connecting the ProActive OutputGenerator component (*ocomp*) to the running components of the application running in MOCCA and wrapped as a composite ProActive/MOCCA component (*wcomp*).

the monitoring system. The optimizer prototype called GrAppO was implemented and integrated with the GridSpace engine, supporting short- and medium-sighted optimization modes.

Testing involved measurement of optimization quality from different points of view, including unit tests, integration tests with other components of GridSpace, and finally quality tests to verify the usefulness of GrAppO operations in a given environment. Since connections to the Monitoring System and PROToS were not yet available (the systems were not configured to receive and answer requests), tests were performed using mock components and simulated data. The objective function for the test was minimization of *makespan*, which is a measure of the throughput of a heterogeneous computing system. It is defined as a total completion time for a group of tasks.

Out of several tests which were performed, the one described here aimed to investigate how GrAppO results are influenced by the lack of resource monitoring information. The test involved scheduling operations of 20 Grid Object Classes on 10 resources. It was assumed that monitoring information for some of the Grid Object Classes was missing; in such a case random resource selection was performed – otherwise, the medium-sighted mode with max-min heuristics was used. The test was repeated in 10% steps, with up to 50% of data missing.

As shown in Fig. 9.17, the lack of information from external data sources has significant influence on the makespan. The greater the fraction of unavailable data, the greater the deterioration of the makespan. This relation is, however, non-linear. With 10% of information missing the rate of aggravation of the makespan is approximately commensurate (9.81%), whereas, when half of the information is unreach-

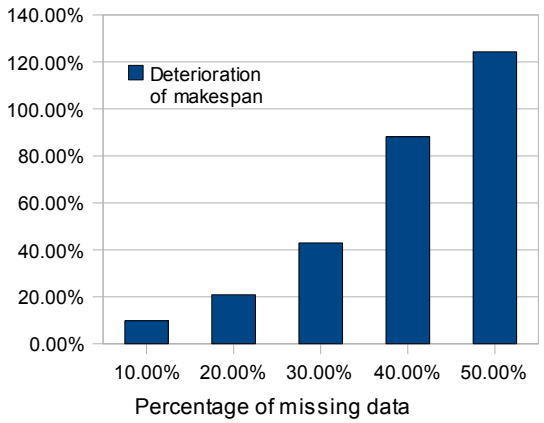


Figure 9.17: Effect of missing data on optimization results

able, the rate reaches 124%. This effect is caused by the necessity of switching to a randomized algorithm.

Although the focus here is not on detailed testing of the GridSpace optimizer module, two important conclusions can be drawn from these tests. First, the experience with the optimizer demonstrates that the high-level scripting approach to application composition can be subject to runtime optimization. The second conclusion is that the monitoring information is crucial for optimization of execution: even a simple heuristic can considerably outperform the random resource selection algorithm. More detailed tests of GrAppO, as well as initial concepts related to using the Askalon workflow scheduling system, are described in [122].

9.11 Conclusions

This chapter presented the experience with applying the component approach to the problem of programming scientific applications. A number of applications of **various levels of coupling** as well as computation and communication requirements were developed and successfully run, including AFC, communication benchmark, gold cluster formation simulation, cellular automata and data mining. All of them demonstrated the advantages of the proposed component-based approach, enabling modularization, **facilitated deployment** and **reconfiguration**.

Experience with the **high-level scripting** approach shows that it can be particularly suitable for rapid development of flexible **scientific experiments**, which is the case in the ViroLab virtual laboratory. Dynamically-deployed and stateful components appeared to be more convenient for modelling such applications as data mining with Weka than other (e.g. service-oriented) approaches. The tests also suc-

cessfully demonstrated the **interoperability** of CCA and GCM component models.

The performance tests of the benchmark applications demonstrated the applicability of the proposed solution to a **wide range of target environments**: from single servers, through clusters to diverse Grid systems such as CrossGrid and Grid'5000 or transatlantic configurations. Experience with **large-scale deployments** allowed the author to obtain insight into the behavior of the environment in such cases and convinced him that that the design and implementation satisfy the stated requirements.

Preliminary performance evaluation of the GrAppO optimization module shows that the proposed programming model can be subject to automatic optimization. This is particularly important for a high-level environment, which follows the stated goals of **facilitating application execution** and relieving the users from low-level details and decisions regarding the underlying computational infrastructure.

The general conclusion from the experiments is that the proposed methodology proved to **effectively support** building and running scientific applications on the described infrastructures. During the course of the work the author had to overcome many problems, such as finding the right design decisions and solving implementation issues. Finally, the results achieved with the existing proof-of-concept implementation of the new solutions justify the adequacy of the proposed methodology.

The results presented in this chapter are partially published: the AFC application sample and communication benchmarks (Sections 9.2 and 9.6) were published in [123]; the gold cluster application (Section 9.3) was described in [29, 119]; Weka experiments (Section 9.4) were outlined in [15]; interoperability experiments (Section 9.9) were the subject of [118, 120]; and optimizer tests (Section 9.10) were published in [122].

Chapter 10

Conclusions and Future Work

In this chapter the author summarizes the contribution of the work, namely the new methodology of combining CCA and H2O models together with higher-level tools which support programming and running scientific applications on the Grid. The achievement of research objectives and the validity of the proposed thesis is confirmed. Subsequently, some more general conclusions are drawn and the prospects for interesting future work, identified in the course of this research, are presented.

10.1 Summary of the contribution

In this work the author analyzed the problem of programming complex scientific applications on the Grid infrastructure. Since this problem remains an important challenge, a new methodology was proposed and paired with a programming and execution environment. The methodology is based on a component programming model, supported by a virtualization mechanism which is adjusted to the Grid infrastructure and enhanced with a set of tools facilitating programming applications on a higher level of abstraction. The component model can be used as a basis for the proposed methodology, since it allows flexible composition (in space and in time), and supports deployment of component code, by using the concept of lightweight containers. It possesses adaptive capabilities and facilitates interoperability between multiple programming languages and other Grid systems.

In order to focus on concrete solutions, the author selected CCA as the underlying component model, and H2O as the virtualization platform for shared resources. By combining matching concepts of CCA and H2O, the author showed that it is possible to develop a programming environment which satisfies the requirements of e-Science applications and is capable of exploiting the Grid infrastructure.

As a result, the research demonstrated that the proposed methodology can fulfill the following requirements:

- **Facilitating high-level programming** is supported by offering a high-level

scripting environment or declaratively using an Architecture Description Language approach.

- **Facilitating deployment on shared resources** of (possibly custom) component code is possible thanks to H2O dynamic deployment mechanisms. When utilizing existing infrastructures, such as EGEE, a dynamically-managed pool of component containers can be created.
- **Scalable to diverse environments** – this feature was demonstrated by deploying the test applications on a wide range of resources: from single laptops, through clusters, to national and international Grid testbeds.
- **Communication adjusted to various levels of coupling** – By applying the RMIX communication library, inter-component bindings can use protocols adjusted to the environment, from local in-process connections to P2P overlay networks using JXTA.
- **Supporting multiple programming languages** – Having chosen the CCA model, it is possible to use the Babel tool for providing interoperability between multiple programming languages, including Java, C and Fortran. The feasibility of such an approach was examined by integrating Babel with RMIX.
- **Adapted to the unreliable Grid environment** – This is achieved by combining dynamic capabilities of the CCA model and the H2O platform, which together enable creation of the environment that can manage and reconfigure applications at runtime to reflect changes in the environment.
- **Interoperability** with other component models was demonstrated on the example of the Grid Component Model and the ProActive framework.

The author showed the feasibility of creating such a programming environment via development of prototype solutions, demonstrating how each of the desired features can be implemented. The resulting solution was validated with a number of testing and real-life scientific applications. The conducted experiments validated various aspects of the proposed methodology.

The innovative aspects of this research work can be summarized as follows:

- A new component-based approach of combining CCA and H2O models.
- New high-level scripting- and ADL-based approaches to component composition.
- A new CCA framework for metacomputing and Grid applications (MOCCA).

- New solutions for interoperability between CCA and GCM component models, combining Babel and RMIX.
- A new method of running component-based applications on the Grid and other distributed resources, combined with a peer-to-peer communication system.

Results of the research and development work described in Chapters 4–9 prove the validity of the *thesis* formulated in Sec. 1.4. It was shown that the proposed methodology, based on the CCA component model and combined with the virtualization layer of H2O as well as with higher-level programming tools, can effectively support programming and execution of scientific applications on the Grid. Therefore, the research objectives of the thesis are successfully achieved, including the concepts, methods, tools and feasibility studies. The author is convinced that the proposed solutions constitute an interesting and valuable contribution to the field of computer science.

10.2 Conclusions and discussion

The main conclusion from the conducted research is that choosing a component model and a lightweight resource sharing model is an appropriate solution. The selection of CCA and H2O as sample technologies was motivated by pragmatic reasons, since both provide tools which facilitate development and demonstration of the prototype programming environment. Nevertheless, it is important to note that both the model and the platform are general in scope and it is possible to use other technologies than CCA and H2O. This was demonstrated in Chapter 4 by offering high-level application composition based on a scripting approach, which is technology-neutral. Moreover, Chapter 7 shows that it is possible to combine components from many models and frameworks into one application, thus hiding the details of any specific component standard.

The author tried to ensure that the concepts and methods devised in this thesis are of a general nature and can thus outlive specific technologies and the implementations. This is especially important in the world of rapidly-changing Grid and Web middleware solutions. Experience gained from experiments on constructing applications from components and providing higher-level tools and abstractions will be useful even when new underlying technologies emerge. Moreover, the methods of creating virtualization layers over heterogeneous resources will gain importance as increasingly greater numbers of resource and device types become available for solving computational problems, ranging from petascale supercomputers, through gaming consoles such as PlayStation, to mobile devices.

One more general remark should be mentioned here in relation to the model of communication between components. In CCA, interactions are limited to RPC-style

invocations: a component with a *uses* port can invoke methods on the connected *provides* port. This implies a synchronous request-response model. However, some component models support asynchronous interactions directly, either as an event system (as in CCM) or as asynchronous RMI (as in GCM and its implementation in ProActive). Other types of interactions include streaming and more complex application protocols, such as those supported by BEEP [152]. The author's experience shows that the simple RPC model of interactions is not always sufficient or convenient for many classes of applications, hence work on supporting new types of component ports remains important. Nevertheless, it should be noted that this issue emerges on the level of the base component model, while all higher-level tools for component composition proposed in this thesis remain valid and usable.

Another important conclusion is that scientific applications may be very diverse and include a broad range of possible scenarios. Therefore, it is hardly possible to propose a single programming model which could cover all of them. It was shown that the component model can be regarded as one of the most promising models when tackling these scenarios. To make it usable, however, a wide range of high-level tools needs to be provided which should be complementary in their roles. Examples include scripting and ADL (descriptor-based) approaches, described in Chapters 4 and 5, which constitute alternative solutions to the component composition problem. The development of such models and tools remains a highly relevant research challenge.

10.3 Future work

Although the research and development work performed and described in this thesis has successfully concluded and the objectives defined in Sec. 1.4 have been achieved, the author cannot state that the work is fully complete. To the contrary, the topic of programming applications on the Grid is so wide, that there is always room for further investigations and improvements. Below is a list of future research directions which were identified during the work on this thesis.

Systematic development of supporting algorithms This thesis describes a top-down approach to creating a programming environment, defining its behavior and usage from the programmer's point of view. The assumption is that the programmer should be unaware of the underlying algorithms supporting the automatic capabilities of the environment. When developing the prototypes, focus was on providing a high-level APIs (script) and ADL specifications, along with simple algorithms, e.g. for deployment planning. A wide range of interesting research issues is related to the development and improvement of specific algorithms and policies. These, however, should be pursued in response to more concrete application and

infrastructure requirements.

Higher-level programming support using semantic Web concepts It would be valuable to enhance the methodology with the recent achievements of Semantic Grid (Web) technologies. Using e.g. ontologies to provide semantic descriptions and reasoning for components would facilitate the process of component (service) discovery, matchmaking and interoperability. For instance, techniques used by semantic Web services and the knowledge Grid could be adapted to component environments without much effort. Moreover, the semantic framework developed in the context of the GridSpace model could be reused here.

Development of a more integrated environment The current prototype of the programming environment, as described in this thesis, has the form of a number of tools and solutions which are centered around the same problem and solve its individual aspects. Still, to make the environment more usable (in addition to improvement of software engineering quality, bugfixes, etc.), it would be worthwhile to provide a more tightly integrated, component-based programming toolkit, enhanced with a friendly development environment, e.g. based on the Eclipse platform.

Development of a formal model The formal aspect of this thesis was limited to the usage of component model specifications (CCA, Fractal), ADL definitions (design of ADLM), scripting API design and usage of UML notation where appropriate. In addition to these, a more formal approach could be derived, which would enable reasoning about the properties of the environment and the application, model checking etc. Such a formal approach is developed for the Fractal component model and it may therefore be possible to extend it towards the CCA model, which is applied in this thesis.

GCM interoperability enhancement In this area, future work may focus on automatic ADL building, generation of glue components at runtime and investigating advanced features by which GCM extends the Fractal model. These need to be compared to their counterparts which are partially present, although not standardized in CCA frameworks. An interesting extension of the current work would be integration with the Babel system, which provides programming language interoperability for CCA. More performance tests to measure the overhead introduced by the glue layer could also give hints related to the applicability of the proposed approach in comparison to other solutions, such as Web services.

Full support for multilanguage components Experience with the Babel tool described in Chapter 7 yields prospects for full integration of MOCCA with Ba-

bel and for supporting native CCA components in the framework. This, however, requires much low-level development work and is therefore left out of the scope of this thesis. It will be a crucial feature to enable the environment to be practically applicable in more real-life applications.

Security support Security should be regarded as an aspect orthogonal to the programming model, and should be transparent to the application programmer. It should, however, be carefully integrated with the environment in a suitable way. The security aspect was intentionally left out of the scope of this thesis; nevertheless, it may be noted here that the H2O platform offers security mechanisms which are adequate to the resource sharing model it supports. On the other hand, existing Grid infrastructures rely on such security solutions as GSI or Shibboleth. Therefore, the author has initiated research and development work on the integration of the H2O platform with these security mechanisms.

Abbreviations and Acronyms

Abbreviation	Explanation
ADL	Architecture Description Language
ADLM	ADL for MOCCAccino
AFC	Application Flow Composer [26, 27]
AOM	Application Object Model
API	Application Programming Interface
ASSIST	A Software development System based upon Integrated Skeleton Technology [1]
ATLAS	A Toroidal LHC ApparatuS [79]
BEEP	Blocks Extensible Exchange Protocol
BPEL	Business Process Execution Language (for Web Services) [137]
CCA	Common Component Architecture [36, 9, 4]
CCAFFEINE	Component framework implementing CCA standard [4]
CCM	CORBA Component Model [139]
CERN	European Laboratory for Particle Physics
CORBA	Common Object Request Broker Architecture [138]
DAG	Directed Acyclic Graph
DCA	Distributed CCA Framework based on MPI [19]
DCOM	Distributed Component Object Model [42]
DHT	Distributed Hash Table
DG-ADAJ	Desktop Grid Adaptive Distribution of Applications in Java [141, 6]
CPU	Central Processing Unit
DEISA	Distributed European Infrastructure for Supercomputing Applications [44]
EC2	Elastic Computing Cloud [7]
EBI	European Bioinformatics Institute
EGEE	Enabling Grids for E-SciencE [48]
ETSI	European Telecommunications Standards Institute
EJB	Enterprise Java Beans [50]
FFT	Fast Fourier Transform

FTP	File Transfer Protocol
GAT	Grid Application Toolkit [5]
GC	Garbage Collector
GCM	Grid Component Model [40, 65]
GIOP	General Inter-ORB Protocol
gLite	Lightweight Middleware for Grid Computing [49]
GOI	Grid Opearation Invoker [15]
GOS	Grid Operating System
GrADS	Grid Application Development Software [17]
GRAM	Grid Resource Allocation and Management
GrAppO	GridSpace Application Optimizer
GridCCM	Grid CORBA Component Model extenstions [148]
GRR	Grid Resource Registry [28]
GSFL	Grid Service Flow Language [107]
GT4	Globus Toolkit 4
GUI	Graphical User Interface
H2O	H2O Resource Sharing Platform [108]
HDNS	Harness Distributed Name Service [72]
HEP	High Energy Physics
HIV	Human Immunodeficiency Virus
HLA	High Level Architecture [86]
HOC-SA	Higher-Order Components-Service Architecture [46]
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
ICENI	Imperial College e-Science Networked Infrastructure [128]
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
INRIA	Institut National de Recherche en Informatique et en Automatique
IOR	Intermediate Object Representation
IT	Information Technology
J2EE	Java 2 Enterprise Edition
JAR	Java archive file
JDL	Job Description Language
JMS	Java Messaging System
JNI	Java Native Interface
JRMP	Java Remote Method Protocol
JSDL	Job Submission Description Language
JVM	Java Virtual Machine
JXTA	JXTA Peer-to-Peer technoloy [99]
KIP	Kernel Information Provider

K-WfGrid	Knowledge-based Workflow System for Grid Applications [80]
L-BFGS	Limited-memory Broyden-Fletcher-Goldfarb-Shanno
LAN	Local Area Network
LCG	LHC Computing Grid
LCF	LCG File Catalog
LDAP	Lightweight Directory Access Protocol
LHC	Large Hadron Collider
LSF	Load Sharing Facility
MDS	Monitoring and Discovery Service
MOCCA	Distributed CCA Framework for Metacomputing
MPI	Message Passing Interface [132]
MPICH	Implementation of MPI standard [101]
N/A	Not Available
NAREGI	National Research Grid Initiative [130]
NAT	Network Address Translation
NGS	National Grid Service
NS	Name Service
NSF	National Science Foundation
OGF	Open Grid Forum
OGSA	Open Grid Services Architecture [56]
OGSI	Open Grid Services Infrastructure
OMG	Object Managment Group
OSG	Open Science Grid [142]
P2P	Peer to Peer
PB	Petabyte
PBS	Portable Batch System [143]
PC	Personal Computer
PROToS	Provenance Tracking System [13]
PVM	Parallel Virtual Machine [166]
RAD	Rapid Application Development
RMI	Remote Method Invocation [165]
RMIX	Multiprotocol RMI library [109]
ROOT	An Object Oriented Framework For Large Scale Data Analysis [23]
RPC	Remote Procedure Call
SCA	Service Component Architecture [14]
SCIRun	A Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI) [192]
SIDL	Scientific Interface Definition Language [104]
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol 1.1 [189]

SOKU	Service Oriented Knowledge Utilities
SSH	Secure Shell
SWIG	Simplified Wrapper and Interface Generator [167]
TAR	Tape archive file
TCP	Transmission Control Protocol
UNICORE	Uniform Interface to Computing Resources [179]
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
US	United States
VDT	Virtual Data Toolkit
ViroLab	A Virtual Laboratory for Decision Support in Viral Disease Treatment [184, 185, 160]
WAN	Wide Area Network
WS	Web Service
WS-GRAM	Web Service-based Grid Resource Allocation and Management
WSDL	Web Services Description Language [188]
WSRF	Web Services Resource Framework [190]
XCAT	XML-based CCA Toolkit [106]
XML	eXtensible Markup Language
XSOAP	Implementation of SOAP Protocol [75]

Bibliography

[1] M. Aldinucci, S. Campa, M. Coppola, M. Danelutto, D. Laforenza, D. Puppin, L. Scarponi, M. Vanneschi, and C. Zoccolo. Components for High Performance Grid Programming in Grid.IT. In V. Getov and T. Kielmann, editors, *Proceedings of the Workshop on Component Models and Systems for Grid Applications, ICS '04 Intl. Conf.*, CoreGRID, pages 19–38. Springer, 2005.

[2] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in gcm: Autonomic management of grid components. In *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 13-15 February 2008, Toulouse, France*, pages 54–63. IEEE Computer Society, 2008.

[3] M. Aldinucci, M. Danelutto, A. Paternesi, R. Ravazzolo, and M. Vanneschi. Building interoperable grid-aware ASSIST applications via WebServices. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. G. Plata, P. Tirado, and E. L. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain*, volume 33 of *John von Neumann Institute for Computing Series*, pages 145–152. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[4] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.

[5] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schott, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards generic and easy application programming interfaces for the grid. *Proceedings of the IEEE*, 93(3):534–550, Mar. 2005.

[6] I. Alshabani, R. Olejnik, B. Toursel, M. Tudruj, and E. Laskowski. A framework for desktop grid applications: CCADAJ. In *5th International Symposium on Parallel and Distributed Computing (ISPDC 2006), 6-9 July 2006, Timisoara, Romania*, pages 208–214, 2006.

[7] Amazon.com. Elastic compute cloud (EC2), 2008. `aws.amazon.com/ec2`.

[8] G. Antoniu, M. Jan, and D. A. Noblet. Enabling the P2P JXTA platform for high-performance networking grid infrastructures. In L. T. Yang, O. F. Rana, B. D. Martino, and J. Dongarra, editors, *High Performance Computing and Communications, First International Conference, HPCC 2005, Sorrento, Italy, September 21-23, 2005, Proceedings*, volume 3726 of *Lecture Notes in Computer Science*, pages 429–439. Springer, 2005.

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

[10] A. M. Artoli, D. Kandhai, H. C. J. Hoefsloot, A. G. Hoekstra, and P. M. A. Sloot. Lattice bgk simulations of flow in a symmetric bifurcation. *Future Gener. Comput. Syst.*, 20(6):909–916, 2004.

[11] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. Programming, deploying, composing, for the grid. In J. C. Cunha and O. F. Rana, editors, *Grid Computing: Software Environments and Tools*. Springer, January 2006.

[12] B. Baliś, M. Bubak, W. Funika, R. Wismüller, M. Radecki, T. Szepieniec, T. Arodź, and M. Kurdziel. Performance evaluation and monitoring of interactive grid applications. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, September 19-22, 2004, Proceedings*, volume 3241 of *Lecture Notes in Computer Science*, pages 345–352. Springer, 2004.

[13] B. Baliś, M. Bubak, and J. Wach. Provenance Tracking in the ViroLab Virtual Laboratory. In R. Wyrzykowski, editor, *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 2007, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2007. to appear.

[14] G. Barber. Service component architecture home, 2007. `http://osoa.org/display/Main/Service+Component+Architecture+Home`.

[15] T. Bartyński, M. Malawski, T. Gubala, and M. Bubak. Universal grid client: Grid operation invoker. In R. Wyrzykowski, editor, *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007, Gdansk, Poland, September 2007, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2007. to appear.

[16] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[17] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. M. Crummey, D. A. Reed, L. Torczon, and R. Wolski. The GrADS project: Software support for high-level grid application development. *Intl. Journal of High Performance Computing Applications*, 15(4):327–344, 2001.

[18] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[19] F. Bertrand and R. Bramley. Dca: A distributed cca framework based on mpi. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, pages 80–89. IEEE Computer Society, 2004.

[20] F. Bertrand, R. Bramley, A. Sussman, D. E. Bernholdt, J. A. Kohl, J. W. Larson, and K. Damevski. Data redistribution and remote method invocation in parallel component architectures. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CA, USA*. IEEE Computer Society, 2005.

[21] R. Bjornson, N. Carriero, D. Gelernter, and J. Leichter. Linda in adolescence. In *ACM SIGOPS European Workshop*, 1986.

[22] H. Bouziane, C. Perez, and T. Priol. Combining a software component model and a workflow language into a component model with spatial and temporal compositions. Research Report 6421, INRIA, 2008. `https://hal.inria.fr/inria-00211158`.

[23] R. Brun and F. Rademakers. Root: An object oriented data analysis framework. *Nucl. Instrum. Meth.*, A389:81–86, 1997. See also: `http://root.cern.ch`.

[24] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in Java. *Softw., Pract. Exper.*, 36(11-12):1257–1284, 2006.

[25] M. Bryliński, W. Jurkowski, L. Konieczny, and I. Roterman. Limited conformational space for early-stage protein folding simulation. *Bioinformatics*, 20(2):199–205, 2004.

[26] M. Bubak, K. Górka, T. Gubała, M. Malawski, and K. Zając. Component-based system for grid application workflow composition. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, volume 2840 of *Lecture Notes in Computer Science*, pages 611–618. Springer, 2003.

[27] M. Bubak, T. Gubala, M. Kapalka, M. Malawski, and K. Rycerz. Workflow composer and service registry for grid applications. *Future Generation Computer Systems*, 21(1):79–86, 2005.

[28] M. Bubak, T. Gubala, M. Kasztelnik, M. Malawski, P. Nowakowski, and P. Sloot. Collaborative virtual laboratory for e-health. In P. Cunningham and M. Cunningham, editors, *Expanding the Knowledge Economy: Issues, Applications, Case Studies, eChallenges e-2007 Conference Proceedings*, page 8, Amsterdam, 2007. IOS Press.

[29] M. Bubak, M. Malawski, and M. Placek. Using MOCCA component environment for simulation of gold clusters. In M. Bubak, M. Turała, and K. Wiatr, editors, *Proceedings of Cracow Grid Workshop - CGW'05, November 20-23 2005*, pages 295–299, Krakow, Poland, 2006. ACC-Cyfronet AGH.

[30] A. I. D. Bucur and D. H. J. Epema. Scheduling policies for processor coallocation in multicluster systems. *IEEE Trans. Parallel Distrib. Syst.*, 18(7):958–972, 2007.

[31] E.-K. Byun and J.-S. Kim. Dynagrid: A dynamic service deployment and resource migration framework for WSRF-compliant applications. *Parallel Comput.*, 33(4-5):328–338, 2007.

[32] B. Carlsson and R. Gustavsson. The rise and fall of Napster - an evolutionary approach. In *AMT '01: Proceedings of the 6th International Computer Science Conference on Active Media Technology*, pages 347–354, London, UK, 2001. Springer.

[33] D. Caromel, A. di Costanzo, and C. Delbé. Peer-to-peer and fault-tolerance: Towards deployment-based technical services. *Future Generation Comp. Syst.*, 23(7):879–887, 2007.

[34] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer, 2005.

[35] D. A. Case, T. E. Cheatham III, T. Darden, H. Gohlke, R. Luo, K. M. M. Jr., A. Onufriev, C. Simmerling, B. Wang, and R. J. Woods. The Amber biomolecular simulation programs. *Journal of Computational Chemistry*, 26(16):1668–1688, 2005.

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

[37] K. Chiu, M. Govindaraju, and D. Gannon. The proteus multiprotocol message library. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–9, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[38] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press and Pitman, 1989.

[39] M. Coppola, M. Danelutto, S. Lacour, C. Pérez, T. Priol, N. Tonellotto, and C. Zoccolo. Towards a common deployment model for grid systems. In S. Gorlatch and M. Danelutto, editors, *Proc. of the Integrated Research in Grid Computing Workshop*, volume TR-05-22, pages 31–40, Pisa, Italy, Nov. 2005. Università di Pisa, Dipartimento di Informatica.

[40] CoreGRID Programming Model Virtual Institute. Basic features of the grid component model (assessed), 2006. Deliverable D.PM.04, CoreGRID, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[41] K. Czajkowski, C. Kesselman, S. Fitzgerald, and I. T. Foster. Grid information services for distributed resource sharing. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 2001), 7-9 August 2001, San Francisco, CA, USA*, pages 181–194. IEEE Computer Society, 2001.

[42] COM: Component Object Model Technologies, 2004. `www.microsoft.com/com/default.mspx`.

[43] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing: Second European AcrossGrids Conference, AxGrids*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer, 2004.

[44] DEISA. DEISA project, 1999. http://deisa.org.

[45] A. Denis, C. Pérez, and T. Priol. PadicoTM: an open integration framework for communication middleware and runtimes. *Future Gener. Comput. Syst.*, 19(4):575–585, 2003.

[46] J. Duennweber and S. Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *Services Computing, 2004 IEEE Int. Conf. on (SCC'04)*, pages 288–294, Shanghai, China, 2004. IEEE.

[47] Eclipse Foundation. SOA Tools Platform Project: Service Component Architecture Tools, 2008. http://www.eclipse.org/stp/sca/index.php.

[48] EGEE Project. Website, 2006. http://public.eu-egee.org/.

[49] EGEE Project. Lightweight middleware for grid computing, 2007. http://glite.web.cern.ch/glite/.

[50] Enterprise JavaBeans technology. http://java.sun.com/products/ejb/.

[51] W. R. Elwasif, B. R. Norris, B. A. Allan, and R. C. Armstrong. Bocca: a development environment for HPC components. In *CompFrame '07: Proceedings of the 2007 symposium on Component and framework technology in high-performance and scientific computing*, pages 21–30, New York, NY, USA, 2007. ACM.

[52] I. Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[53] I. Foster. Service-oriented science. *Science*, 308(5723):814 – 817, 2005.

[54] I. Foster, N. T. Karonis, C. Kesselman, G. Koenig, and S. Tuecke. A secure communications infrastructure for high-performance distributed computing. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, page 125, Washington, DC, USA, 1997. IEEE Computer Society.

[55] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan-Kaufmann, 1999.

[56] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002. http://www.globus.org/alliance/publications/papers/ogsa.pdf.

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

[58] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, July 2002.

[59] W. Funika, D. Harężlak, D. Krol, and M. Bubak. Environment for collaborative development and execution of virtual laboratory applications. In M. Bubak, G. Albada, J. Dongarra, and P. Sloot, editors, *Computational Science - ICCS 2008, 8th International Conference Proceedings*, Lecture Notes in Computer Science, Krakow, Poland, June 2008.

[60] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[61] D. Gannon, J. Alameda, O. Chipara, M. Christie, V. Dukle, L. Fang, M. Farrellee, G. Kandaswamy, D. Kodeboyina, S. Krishnan, C. Moad, M. Pierce, B. Plale, A. Rossi, Y. Simmhan, A. Sarangi, A. Slominski, S. Shirasuna, and T. Thomas. Building grid portal applications from a web service component architecture. *Proceedings of the IEEE*, 93(3):551–563, 2005.

[62] D. Gannon, R. Ananthakrishnan, S. Krishnan, M. Govindaraju, L. Ramakrishnan, and A. Slominski. Grid web services and application factories. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 251–264. Wiley, 2003.

[63] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. Programming the Grid: Distributed software components, P2P and Grid web services for scientific applications. *Cluster Computing*, 5(3):325 – 336, Jul 2002.

[64] The official Gausian website, 2007. `http://www.gaussian.com/`.

[65] Deliverable D.PM.02 - proposals for a Grid component model, 2006. `http://www-sop.inria.fr/oasis/Ludovic.Henrio/CoreGrid/GCM-proposal.pdf`.

[66] W. Gentzsch. Grid initiatives: Lessons learned and recommendations. Technical report, RENCI, Duke D-Grid, Jan 2007.

[67] V. Getov and T. Kielmann, editors. *Component Models and Systems for Grid Applications.* Springer, 2005.

[68] The Globus Alliance, 2004. `http://www.globus.org`.

[69] GLUE Project. GLUE information model, 2005. `http://infnforge.cnaf.infn.it/glueinfomodel/`.

[70] J. Gomes, M. David, J. Martins, L. Bernardo, A. García, M. Hardt, H. Kornmayer, J. Marco, R. Marco, D. Rodríguez, I. Diaz, D. Cano, J. Salt, S. Gonzalez, J. Sánchez, F. Fassi, V. Lara, P. Nyczyk, P. Lason, A. Ozieblo, P. Wolniewicz, M. Bluj, K. Nawrocki, A. Padee, W. Wislicki, C. Fernández, J. Fontán, Y. Cotronis, E. Floros, G. Tsouloupas, W. Xing, M. D. Dikaiakos, J. Astalos, B. A. Coghlan, E. Heymann, M. A. Senar, C. Kanellopoulos, A. Ramos, and D. Groen. Experience with the international testbed in the CrossGrid project. In *Advances in Grid Computing - EGC 2005, European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers*, volume 3470 of *Lecture Notes in Computer Science*, pages 98–110. Springer, 2005.

[71] Google.com. Appengine, 2008. `http://code.google.com/appengine/`.

[72] D. Gorissen, P. Wendykier, D. Kurzyniec, and V. Sunderam. Integrating grid information services using JNDI. In *6th IEEE/ACM International Workshop on Grid Computing (Grid 2005)*, Seattle, Washington, USA, Nov. 2005.

[73] M. Govindaraju et al. Design of Distributed Component Frameworks for Computational Grids. In *Proc. of the Int. Conf. on Comm. in Computing (CIC)*, pages 160–166, June 2004.

[74] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. Merging the CCA component model with the OGSI framework. In *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 182, Washington, DC, USA, 2003. IEEE Computer Society.

[75] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *SC*, 2000.

[76] P. Graham, M. Heikkurinen, J. Nabrzyski, A. Oleksiak, , M. Parsons, H. Stockinger, K. Stockinger, M. Stroinski, and J. Węglarz. EU Funded Grid Development in Europe. In M. D. Dikaiakos, editor, *Grid Computing: Second European AcrossGrids Conference, AxGrids 2004, Nicosia, Cyprus, January 28-30, 2004. Revised Papers*, volume 3165 of *Lecture Notes in Computer Science*, pages 1–10. Springer, Jan. 2004.

[77] S. L. Graham, M. Snir, and C. A. Patterson, editors. *Getting Up to Speed, The Future of Supercomputing*. The National Academies Press, 2006.

[78] Grid'5000. Public portal, 2007. `https://www.grid5000.fr/`.

[79] A. H. Group. Atlas high-level trigger, data acquisition and controls technical design report. Technical Report 1.4, CERN, October 2003.

[80] T. Gubala, D. Herezlak, M. Bubak, and M. Malawski. Semantic composition of scientific workflows based on the petri nets formalism. In *Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, page 12, Amsterdam, The Netherlands, 2006. IEEE Computer Society.

[81] T. Gubała and M. Bubak. Gridspace - semantic programming environment for the grid. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers*, volume 3911 of *Lecture Notes in Computer Science*, pages 172–179. Springer, 2006.

[82] T. Gubała and A. Hoheisel. Highly dynamic workflow orchestration for scientific applications. In *CoreGRID Intergation Workshop 2006 (CIW06)*, pages 309–320. ACC CYFRONET AGH, 2006.

[83] D. Harężlak. Multilanguage and multiprotocol interoperability: Babel and RMIX – MSc Thesis, 2006. Institute of Computer Science AGH.

[84] S. Heiler. Semantic interoperability. *ACM Computing Surveys*, 27, 1995.

[85] A. Herraez. Jmol: an open-source Java viewer for chemical structures in 3d, 2008. `http://www.jmol.org/`.

[86] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA), 2004. `http://standards.ieee.org/catalog/olis/compsim.html`.

[87] L. Hluchy, O. Habala, M. Maliska, B. Simo, V. Tran, J. Astalos, and M. Babik. Grid based flood prediction virtual organization. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 4, Washington, DC, USA, 2006. IEEE Computer Society.

[88] A. Hoekstra and P. Sloot. Introducing grid speedup gamma: A scalability metric for parallel applications on the grid. In P. Sloot, A. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, editors, *Advances in Grid Computing - EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 245–249. Springer, February 2005.

[89] Y. Huang, A. Slominski, C. Herath, and D. Gannon. Ws-messenger: A web services-based messaging system for service-oriented grid computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 166–173, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[90] P. J. Hunter, W. W. Li, A. D. McCulloch, and D. Noble. Multiscale modeling: Physiome project standards, tools, and databases. *Computer*, 39(11):31–39, 2006.

[91] P. Hwang, D. Kurzyniec, and V. Sunderam. Heterogeneous parallel computing across multidomain clusters. In *Proc. of 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary*, Lecture Notes in Computer Science. Springer, 2004.

[92] A. Iamnitchi and D. Talia. P2p computing and interaction with grids. *Future Generation Comp. Syst.*, 21(3):331–332, 2005.

[93] IBM. What is grid computing, 2007. `http://www-03.ibm.com/grid/about_grid/what_is.shtml`.

[94] K. Iskra, R. G. Belleman, G. D. van Albada, J. Santoso, P. M. A. Sloot, H. E. Bal, H. J. W. Spoelder, and M. Bubak. The polder computing environment: a system for interactive distributed simulation. *Concurrency and Computation: Practice and Experience*, 14(13-15):1313–1335, 2002.

[95] W. M. Jones, W. B. L. III, L. W. Pang, and D. C. S. Jr. Characterization of bandwidth-aware meta-schedulers for co-allocating jobs across multiple clusters. *The Journal of Supercomputing*, 34(2):135–163, 2005.

[96] Java powered Ruby implementation, 2007. `http://jruby.codehaus.org/`.

[97] M. Jurczyk, P. Golenia, M. Malawski, D. Kurzyniec, M. Bubak, and V. S. Sunderam. A system for distributed computing based on H2O and JXTA. In M. Bubak, M. Turała, and K. Wiatr, editors, *Proceedings of Cracow Grid Workshop - CGW'04, December 13-15 2004*, pages 257–268, Kraków, 2005. ACC-Cyfronet AGH.

[98] P. Jurczyk, M. Golenia, M. Malawski, D. Kurzyniec, M. Bubak, and V. S. Sunderam. Enabling remote method invocations in peer-to-peer environments: RMIX over JXTA. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers*, volume 3911 of *Lecture Notes in Computer Science*, pages 667–674. Springer, 2006.

[99] The JXTA Project, 2004. `http://www.jxta.org`.

[100] The Jython Website, 2004. `http://www.jython.org`.

[101] N. T. Karonis, B. R. Toonen, and I. T. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *J. Parallel Distrib. Comput.*, 63(5):551–563, 2003.

[102] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual workspaces: Achieving quality of service and quality of life in the grid. *Scientific Programming Journal*, 13(4):265–276, 2005.

[103] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[104] S. R. Kohn et al. Divorcing Language Dependencies from a Scientific Software Library. In *Proc. of the 10th SIAM Conf. on Parallel Processing for Sci. Comp.*, Portsmouth, USA, Mar. 2001. SIAM.

[105] S. Krishnan and D. Gannon. Checkpoint and restart for distributed components in xcat3. In R. Buyya, editor, *GRID*, pages 281–288. IEEE Computer Society, 2004.

[106] S. Krishnan and D. Gannon. XCAT3: A Framework for CCA Components as OGSA Services. In *Proc. Int. Workshop on High-Level Parallel Progr. Models and Supportive Environments (HIPS)*, pages 90–97, Santa Fe, New Mexico, USA, Apr. 2004.

[107] S. Krishnan, P. Wagstrom, and G. von Laszewski. Gsfl: A workflow framework for grid services. Technical report, Globus Alliance, 2002. `http://www.globus.org/cog/papers/gsfl-paper.pdf`.

[108] D. Kurzyniec et al. Towards Self-Organizing Distributed Computing Frameworks: The H2O Approach. *Parallel Processing Lett.*, 13(2):273–290, 2003.

[109] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slomiński. RMIX: A multiprotocol RMI framework for java. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, pages 140–146, Nice, France, Apr. 2003. IEEE Computer Society.

[110] A. Labarga, F. Valentin, M. Anderson, and R. Lopez. Web services at the European bioinformatics institute. *Nucleic Acids Res*, 35(Web Server issue), July 2007.

[111] S. Lacour et al. Deploying CORBA components on a computational grid. In W. Emmerich et al., editors, *Component Deployment: 2nd Int. Working Conf., CD 2004, Edinburgh, UK, Proc.*, volume 3083 of *Lecture Notes in Computer Science*, pages 35 – 49. Springer, 2004.

[112] D. Laforenza. Grid programming: some indications where we are headed. 28:1733–1752, 2002.

[113] M. Lewis and A. Grimshaw. The core Legion object model. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 551, Washington, DC, USA, 1996. IEEE Computer Society.

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

[115] M. Malawski, T. Bartyński, and M. Bubak. A tool for building collaborative applications by invocation of grid operations. In M. Bubak, G. Albada, J. Dongarra, and P. Sloot, editors, *Computational Science - ICCS 2008, 8th International Conference Proceedings*, Lecture Notes in Computer Science, Krakow, Poland, June 2008.

[116] M. Malawski, T. Bartyński, E. Ciepiela, J. Kocot, P. Pelczar, and M. Bubak. An ADL-based support for CCA components on the Grid. In *CoreGRID Workshop on Grid Systems, Tools and Environments in Conjunction with GRIDS@work: CoreGRID Conference, Grid Plugtests and Contest*, Sophia-Antipolis, France, December 2006.

[117] M. Malawski, T. Bartyński, E. Ciepiela, J. Kocot, P. Pelczar, and M. Bubak. A new approach to supporting component applications on grid. In M. Bubak,

M. Turała, and K. Wiatr, editors, *Proceedings of Cracow Grid Workshop - CGW'06, October 15 - 18, 2006*, pages 328–336, Krakow, Poland, 2007. ACC-Cyfronet AGH.

[118] M. Malawski, M. Bubak, F. Baude, D. Caromel, L. Henrio, and M. Morel. Interoperability of grid component models: GCM and CCA case study. In *Towards Next Generation Grids, proceedings of the CoreGRID Symposium in conjunction with Euro-Par 2007*, CoreGRID series, pages 95–106. Springer, August 2007.

[119] M. Malawski, M. Bubak, M. Placek, D. Kurzyniec, and V. Sunderam. Experiments with distributed component computing across grid boundaries. In *Proceedings of the HPC-GECO/CompFrame workshop in conjunction with HPDC 2006*, Paris, France, 2006.

[120] M. Malawski, T. Gubala, M. Kasztelnik, T. Bartyński, M. Bubak, F. Baude, and L. Henrio. High-level scripting approach for building component-based applications on the grid. In *CoreGRID Workshop on Grid Programming Model Grid and P2P Systems Architecture Grid Systems, Tools and Environments*, Heraklion, Crete, June 2007. Springer.

[121] M. Malawski, D. Harężlak, and M. Bubak. Towards multiprotocol and multilanguage interoperability: Experiments with Babel and RMIX. In M. Bubak, M. Turała, and K. Wiatr, editors, *Proceedings of Cracow Grid Workshop - CGW'05, November 20-23 2005*, pages 266–278, Krakow, Poland, 2006. ACC-Cyfronet AGH.

[122] M. Malawski, J. Kocot, I. Ryszka, M. Bubak, M. Wieczorek, and T. Fahringer. Optimization of application execution in the gridspace environment. In S. Gorlatch, P. Fragopoulou, and T. Priol, editors, *CoreGRID Integration Workshop 2008 – Integrated Research in Grid Computing*, pages 395–405, April 2008.

[123] M. Malawski, D. Kurzyniec, and V. Sunderam. MOCCA – towards a distributed CCA framework for metacomputing. In *Proceedings of the 10th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS2005) in conjunction with International Parallel and Distributed Processing Symposium (IPDPS 2005)*. IEEE Computer Society, 2005.

[124] M. Malawski, T. Szepieniec, M. Kochanczyk, M. Piwowar, and I. Roterman-Konieczna. The quest for pharmacology active never born proteins within the EUChinaGRID project. In M. Bubak, M. Turała, and K. Wiatr, editors,

Proceegings of the 6th Cracow Grid Workshop, pages 505–510, Krakow, Poland, 2007. ACC CYFRONET-AGH.

[125] Q. H. Mamoud. Getting started with JavaSpaces technology: Beyond conventional distributed programming paradigms, July 2005. `http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/`.

[126] D. L. Martin, M. Paolucci, S. A. McIlraith, M. H. Burstein, D. V. McDermott, D. L. McGuinness, B. Parsia, T. R. Payne, M. Sabou, M. Solanki, N. Srinivasan, and K. P. Sycara. Bringing semantics to web services: The owl-s approach. In J. Cardoso and A. P. Sheth, editors, *Semantic Web Services and Web Process Composition, First International Workshop, SWSWPC 2004, San Diego, CA, USA, July 6, 2004, Revised Selected Papers*, volume 3387 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2004.

[127] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(5-6):817–840, 2004.

[128] A. Mayer, S. McGough, N. Furmento, W. Lee, S. Newhouse, and J. Darlington. ICENI Dataflow and Workflow: Composition and Scheduling in Space and Time. In *UK e-Science All Hands Meeting*, pages 627–634, Nottingham, UK, Sept. 2003. ISBN 1-904425-11-9.

[129] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[130] K. MIURA. Overview of japanese science grid project NAREGI. *Progress in Informatics*, 3:67–75, April 2006.

[131] J. Moscicki, H. Lee, S. Guatelli, S. Lin, and M. Pia. Biomedical applications on the GRID: efficient management of parallel jobs. *Nuclear Science Symposium Conference Record*, 4:2143 – 2147, Oct 2004.

[132] Message Passing Interface Forum. `http://www.mpi-forum.org`.

[133] NASA. Java astrodynamics toolkit (JAT), 2002. `http://opensource.gsfc.nasa.gov/projects/JAT/JAT.php`.

[134] A. Natrajan et al. The Legion Grid Portal. *Concurrency Computat.*, 14(13–15):1365–1394, Nov./Dec. 2002.

[135] C. Nester, M. Philippsen, and B. Haumacher. A more efficient RMI for Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 152–159, New York, NY, USA, 1999. ACM Press.

[136] NGG Group. Next Generation Grids 2 requirements and options for European Grids research 2005-2010 and beyond. Technical report, July 2004.

[137] OASIS WSBPEL Technical Committee. Web services business process execution language version 2.0, Apr. 2007. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`.

[138] Object Management Group, Inc. Common request broker architecture specification, version 3.0.3. Available at: `http://www.omg.org/cgi-bin/doc?formal/04-03-01`, 03 2004.

[139] Object Management Group, Inc. CORBA Component Model, v4.0, 2006. `http://www.omg.org/technology/documents/formal/components.htm`.

[140] Object Management Group, Inc. Deployment and configuration of component-based distributed applications specification, version 4.0, Apr 2006. `http://www.omg.org/docs/formal/06-04-02.pdf`.

[141] R. Olejnik, B. Toursel, M. Tudruj, and E. Laskowski. Optimized java computing as an application for desktop grid. In *Proceedings of the 4th Cracow Grid Workshop*, Krakow, Poland, 2005. ACC CYFRONET-AGH.

[142] Open Science Grid project. Public portal, 2007. `http://www.opensciencegrid.org/`.

[143] OpenPBS. Portable batch system. `http://www.openpbs.org/`.

[144] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[145] A. Padee, W. Padee, and K. Zaremba. Large-scale evolutionary optimization on the grid: Multiple-deme genetic algorithm in the globus-based environment. In M. Bubak, M. Turała, and K. Wiatr, editors, *Proceegings of the 6th Cracow Grid Workshop*, pages 199–206, Krakow, Poland, 2007. ACC CYFRONET-AGH.

[146] S. Parker, K. Zhang, K. Damevski, and C. Johnson. Integrating component-based scientific computing software. In P. R. M.A. Heroux and H. Simon, editors, *Frontiers of Parallel Processing For Scientific Computing*. SIAM, 2005.

[147] N. Parlavantzas, M. Morel, V. Getov, F. Baude, and D. Caromel. Performance and scalability of a component-based grid application. In *9th Int. Worshop on Java for Parallel and Distributed Computing, in conjunction with the IEEE IPDPS conference*, April 2007.

[148] C. Perez, T. Priol, and A. Ribes. A parallel CORBA component model for numerical code coupling. *The International Journal of High Performance Computing Applications (IJHPCA)*, 17(4):417–429, 2003. Special issue Best Applications Papers from the 3rd Intl. Workshop on Grid Computing.

[149] B. Plale, D. Gannon, J. Brotzge, K. Droegemeier, J. Kurose, D. McLaughlin, R. Wilhelmson, S. Graves, M. Ramamurthy, R. D. Clark, S. Yalda, D. A. Reed, E. Joseph, and V. Chandrasekar. Casa and lead: Adaptive cyberinfrastructure for real-time multiscale weather forecasting. *Computer*, 39(11):56–64, 2006.

[150] M. Riedel and D. Mallmann. Standardization processes of the UNICORE grid system. In *Proceedings of 1st Austrian Grid Symposium 2005*, pages 191–203, Schloss Hagenberg, Austria, 2005. Austrian Computer Society.

[151] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In R. L. Graham and N. Shahmehri, editors, *1st International Conference on Peer-to-Peer Computing (P2P 2001), 27-29 August 2001, Linköping, Sweden*, pages 99–100. IEEE Computer Society, 2001.

[152] M. Rose. The blocks extensible exchange protocol core, 2001. RFC 3080.

[153] The Ruby programming language, 2007. `http://www.ruby-lang.org`.

[154] K. Rycerz, M. Bubak, M. Malawski, and P. M. A. Sloot. A grid service for management of multiple HLA federate processes. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005, Poznan, Poland, September 11-14, 2005, Revised Selected Papers*, volume 3911 of *Lecture Notes in Computer Science*, pages 699–706. Springer, 2005.

[155] K. Rycerz, A. Tirado-Ramos, A. Gualandris, S. F. P. Zwart, M. Bubak, and P. M. A. Sloot. Interactive n-body simulations on the grid: HLA versus MPI. *Int. J. High Perform. Comput. Appl.*, 21(2):210–221, 2007.

[156] A. Savva et al. Job submission description language (JSDL) specification, version 1.0. `http://www.ogf.org/documents/GFD.56.pdf`, Nov 2005. Open Grid Forum.

[157] R. Schmidt, S. Benkner, I. Brandic, and G. Engelbrecht. Component based applications programming within a service-oriented grid environment. In *Proceedings of the Workshop on Component Models and Frameworks in High Performance Computing (Compframe 2005)*, Atlanta, GA, USA, June 2005.

[158] J. M. Schopf and J. Węglarz. *Grid Resource Management. State of the Art and Future Trends*, chapter Ten Actions When Grid Scheduling. Kluwer Academic Publishers, 2003.

[159] R. Sirvent, J. M. Perez, R. Badia, and J. Labarta. Automatic grid workflow based on imperative programming languages. *Concurrency and Computation: Practice and Experience*, 18:1169–1186, 2005.

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

[161] P. M. A. Sloot. Preface. In *Computational Science - ICCS 2002, International Conference Proceedings*, volume 2329 of *Lecture Notes in Computer Science*, pages V–VI, Amsterdam, The Netherlands, April 2002. Springer.

[162] I. Sommerville. *Software Engineering, 7th edition.* Pearson Education, 2004.

[163] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[164] G. Stuer et al. Towards OGSA Compatibility in Alternative Metacomputing Frameworks. In M. Bubak et al., editors, *Computational Science - ICCS 2004. 4th Int. Conf., Kraków, Poland, June 2004, Part I*, volume 3036 of *Lecture Notes in Computer Science*, pages 51–58. Springer, 2004.

[165] Sun Developer Network. RMI remote method invocation. `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`.

[166] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency - Practice and Experience*, 2(4):315–339, 1990.

[167] SWIG Project. Simplified wrapper and interface generator, 2007. `http://www.swig.org`.

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

[169] B. A. Tate. *Beyond Java.* O'Reilly, 2005.

[170] I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual grid workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, Sept. 2005.

[171] J. M. Taylor. Defining e-science. `http://www.nesc.ac.uk/nesc/define.html`, 2007. U. K. National e-Science Centre.

[172] TeraGrid project. Public portal, 2007. `http://www.teragrid.org/`.

[173] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[174] R. Thakur and W. Gropp. Open issues in mpi implementation. In L. Choi, Y. Paek, and S. Cho, editors, *Asia-Pacific Computer Systems Architecture Conference*, volume 4697 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2007.

[175] A. Tirado-Ramos and P. M. A. Sloot. A conceptual grid architecture for interactive biomedical applications. In *19th IEEE International Symposium on Computer-Based Medical Systems (CBMS 2006), 22-23 June 2006, Salt Lake City, Utah, USA*, pages 762–767. IEEE Computer Society, 2006.

[176] N. Trebon, A. Morris, J. Ray, S. Shende, and A. D. Malony. Performance modeling of component assemblies. *Concurrency and Computation: Practice and Experience*, 19(5):685–696, 2007.

[177] H. L. Truong, B. Baliś, M. Bubak, J. Dziwisz, T. Fahringer, and A. Hoheisel. Towards distributed monitoring and performance analysis services in the k-wfgrid project. In R. Wyrzykowski, J. Dongarra, N. Meyer, and J. Wasniewski, editors, *PPAM*, volume 3911 of *Lecture Notes in Computer Science*, pages 156–163. Springer, 2005.

[178] UK Open Middleware Infrastructure Institute (OMII). Gridsam - grid job submission and monitoring web service, 2007. `http://gridsam.sourceforge.net/`.

[179] Unicore Forum, 2004. `http://www.unicore.org`.

[180] UniGrids Project. Uniform interface to grid services.

[181] H. J. Vallecillo, A. and J. Troya. Component interoperability. Technical Report ITI-2000-37, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga., 2000.

[182] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. F. H. Hofman, C. J. H. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a flexible and efficient java-based grid programming environment. *Concurrency - Practice and Experience*, 17(7-8):1079–1107, 2005.

[183] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, T. Kielmann, and H. E. Bal. Adaptive load balancing for divide-and-conquer grid applications. *Journal of Supercomputing*, 2006.

[184] ViroLab project. `http://www.virolab.org`.

[185] ViroLab Team. ViroLab Virtual Laboratory, 2007. `http://virolab.cyfronet.pl`.

[186] N. Wilson and R. Johnston. Modelling gold clusters with an empirical many-body potential. *Eur. Phys. J. D*, 12:161–169, 2000.

[187] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, June 2005.

[188] World Wide Web Consortium (W3C). Web services description language (wsdl) 1.1, W3C note, Mar. 2001. `http://www.w3.org/TR/wsdl`.

[189] World Wide Web Consortium (W3C). SOAP version 1.2 W3C recommendation (second edition), Apr. 2007. `http://www.w3.org/TR/soap`.

[190] The WS-Resource Framework, 2004. `http://www.globus.org/wsrf`.

[191] Y. Yan and B. M. Chapman. Comparative study of distributed resource management systems - SGE, LSF, PBS Pro, and LoadLeveler. Technical report, Department of Computer Science, University of Houston, May 2005. `http://www2.cs.uh.edu/~yanyh/pubs/D-RMS-study.pdf`.

[192] K. Zhang et al. SCIRun2: A CCA Framework for High Performance Computing. In *Proc. Int. Workshop on High-Level Parallel Progr. Models and Supportive Environments (HIPS)*, pages 72–79, Santa Fe, New Mexico, USA, Apr. 2004.

[193] K. Zieliński, M. Jarząb, D. Wieczorek, and K. Bałos. JIMS extensions for resource monitoring and management of Solaris 10. In V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *Computational Science - ICCS 2006, 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part IV*, volume 3994 of *Lecture Notes in Computer Science*, pages 1039–1046. Springer, 2006.

Index