THE INTERNATIONAL JOURNAL of HIGH PERFORMANCE COMPUTING APPLICATIONS

Article

The International Journal of High Performance Computing Applications
26(3) 275–295

DOI: 10.1177/1094342011422924
hpc.sagepub.com
SAGE

Component-based approach for programming and running scientific applications on grids and clouds

Maciej Malawski[1,2], Tomasz Gubała[3,4] and Marian Bubak[1,4]

Abstract

This paper presents an approach to programming and running scientific applications on grid and cloud infrastructures based on two principles: the first one is to follow a component-based programming model, the second is to apply a flexible technology which allows for virtualization of the underlying infrastructure. The solutions described in this paper include high-level composition and deployment consisting of a scripting-based environment and a manager system based on an architecture description language (ADL), a dynamically managed pool of component containers, and interoperability with other component models such as Grid Component Model (GCM). We demonstrate how the proposed methodology can be implemented by combining the unique features of the Common Component Architecture (CCA) model together with the H2O resource sharing platform, resulting in the MOCCA component framework. Applications and tests include data mining using the Weka library, Monte Carlo simulation of the formation of clusters of gold atoms, as well as a set of synthetic benchmarks. The conclusion is that the component approach to scientific applications can be successfully applied to both grid and cloud infrastructures.

Keywords

cloud computing, component programming, distributed application, grid computing, programming model

1 Introduction

Recently, such paradigms of scientific investigation as e-Science and *system-level* science have been established (Foster and Kesselman, 2006). E-Science applications 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. Grid infrastructures like EGI (Kranzlmüller et al., 2010), DEISA (Gentzsch et al., 2011), Grid'5000 (Bolze et al., 2006), Open Science Grid (Altunay et al., 2011), and TeraGrid (Beckman, 2005) are now considered the key technological platforms enabling the realization of the e-Science paradigm (Schwiegelshohn et al., 2010). Additionally, there is an evolution from simple computing (metacomputing) infrastructures supporting batch processing to more advanced software systems which provide high-level services. Recently, cloud computing has gained attention from the point of view of scientific applications (Vecchiola et al., 2009; Deelman, 2010). Problems such as access to computation, deployment, and application management still remain a challenge, due to some inherent features of the grid and cloud environments.

The main objective of the research presented in this paper can be stated as follows: *How to program and run e-Science applications on the grid and cloud infrastructures*? Although significant effort is being invested in research on programming models, tools, and environments, the problem remains challenging (NGG Group, 2004). The challenges for scientific application developers and users include limited support for high-level programming models and abstractions, the necessity of deploying application code on shared resources, heterogeneity of the environment in terms of computing nodes and also of the network links between them, lack of single middleware for accessing

[1]AGH University of Science and Technology, Department of Computer Science, Kraków, Poland
[2]Center for Research Computing, University of Notre Dame, Notre Dame, USA
[3]ACC CYFRONET-AGH, Kraków, Poland
[4]Informatics Institute, Universiteit van Amsterdam, Amsterdam, The Netherlands

Corresponding author:
Maciej Malawski, Department of Computer Science, AGH University of Science and Technology, Mickiewicza 30, 30-059 Kraków, Poland
Email: malawski@agh.edu.pl

computing resources, and the fact that the infrastructures are based on diverse concepts and programming models. As an answer to these challenges, we propose a *methodology*, consisting of a set of methods and tools, possibly integrated into a programming environment characterized by the following features:

1. facilitating high-level programming;
2. facilitating deployment on shared resources;
3. scalable to diverse environments;
4. communication adjusted to various levels of coupling;
5. adapted to the unreliable distributed environment;
6. interoperable;
7. secure.

These features have been identified based on our experience with grid computing infrastructures, which pose significant challenges for scientific application developers and users.

Below, the desired features are described in more detail.

Facilitating high-level programming The programming model should allow the composition of the application from smaller blocks (modules) and should be able to express temporal dependencies between them as well as direct connections. This composition should be performable by a third party, not hard-coded in the modules. It should also be possible to compose the application at a high level of abstraction, without the need to specify too many technical, infrastructure- and middleware-specific details.

Facilitating deployment on shared resources The environment should support the deployment, possibly dynamic, of custom application code on the available resource pool, taking into account the heterogeneity of the infrastructure and middleware. That is, it should provide a virtualization layer, capable of hiding the diversity of lower layers.

Scalable to diverse environments 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 or cloud infrastructures. 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 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.

Adapted to the unreliable distributed environment As the environment may be highly dynamic and undependable, it will be crucial for the environment to provide some means of adaptability and fault tolerance.

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. These standards include Web services and, in the case of specific programming models, it will be important to interoperate with their most popular implementations.

Secure Running applications on shared resources requires providing or inter-facing with proper authentication and authorization mechanisms, as well as ensuring isolation between multiple users and their applications. These aspects should be clearly separated from the actual application programming, but be configurable at runtime.

This methodology requires an appropriate *high-level* programming and execution environment based on an appropriate programming model and supported by specific tools and services. An environment supporting these features will simplify the usage of complex computing infrastructures for people involved in e-Science. In this paper, we describe how such an environment can be built following the component-based approach and we explain why we have chosen CCA as a component standard. In order to provide a virtualization layer to the environment, we selected the H2O resource-sharing platform which gives us several benefits thanks to the unique features it offers.

The experience with interactive applications in CrossGrid (Bubak et al., 2003), workflow applications (Bubak et al., 2005; Gubła et al., 2006), and virtual laboratories (Sloot et al., 2006) gave us the opportunity to verify different approaches to constructing such applications. The component-based approach was investigated in various aspects: MOCCA (Malawski et al., 2005) is an implementation of the CCA standard using the H2O platform which provides a lightweight container for components. The GridSpace environment (Malawski et al., 2008a) provides a high-level scripting approach for rapid exploratory programming and it integrates multiple technologies, including services, components, and batch jobs (Malawski et al., 2010). We also conducted experiments with deployment of component applications on grid infrastructures (Malawski et al., 2006b) and the applicability of P2P overlay networks for providing communication in the environment (Jurczyk et al., 2006). Our recent experiments with the Amazon EC2 compute cloud as well as with private clouds (Malawski et al., 2011) indicate that the proposed component-based approach fits the cloud computing model as well.

The main contribution of this paper is to present a complete overview of carefully designed methods and tools which, combined together, support the component-based

approach to e-Science applications development. Interconnected, they constitute a complete and self-sufficient programming and execution environment for scientific applications. We discuss the advantages and disadvantages of the component-based approach based on our experience and the lessons learned.

The paper is organized as follows: after the analysis of the state of the art (Section 2) we underline the advantages of using a component programming model and the rationale for choosing CCA and H2O as base technologies (Section 3). Next, we discuss in detail how all the requirements are met by the methods and tools we develop. In Section 4 we present case studies with model scientific applications, as well as results of benchmarks demonstrating the correctness of the approach taken. Section 5 provides a thorough analysis of our solution, its advantages and shortcomings, and general remarks about lessons learned. Finally, in Section 6 we give a summary and suggestions for future work.

2 State of the art

The programming models which can be used to map computations performed by a program onto the distributed nodes of a grid come from parallel and distributed computing, and include task processing, e.g. PBS (Henderson, 1995), Globus Toolkit (Foster, 2006); message passing, MPICH-G2 (Karonis et al., 2003), OpenMPI (Gabriel et al., 2004); distributed objects, including active objects as in ProActive (Baduel et al., 2006); tuple spaces, including JavaSpaces (Bishop and Warren, 2003) or HLA (HLA, 2010); and component- or service-oriented models. Task processing models require the use of many low-level techniques such as scripting and system tools to build and run their applications. For message passing, the lack of support for application deployment in the programming model, and no mechanisms for high-level composition, remain drawbacks of MPI. The main drawback of distributed object systems such as CORBA (Object Management Group, Inc., 2004) is the tight coupling between objects in terms of dependencies, which becomes an obstacle to the adaptability and flexibility of applications. On the other hand, the component and service-oriented models provide better support for third-party composition and reconfiguration of applications. Generally, components are considered to be larger units of composition than objects (Mougin and Barriolade, 2001), while services are more suitable for loosely-coupled applications based on document exchange (Vogels, 2003), where efficient communication is not essential (Henning, 2008). This makes distributed components an attractive technology for scientific applications which require both performance and flexibility.

At a high level, a programming model defines how the whole application can be *composed* from basic blocks to provide the functionality required by the users. One deals with *composition in space* when there are many application units running (possibly in parallel) and they need to interact with one another using direct links. In component-based systems, there are several techniques of composition: low-level API as in CCA (Armstrong et al., 2006), scripting languages, descriptor-based programming – ADL as in Fractal (Bruneton et al., 2006), skeletons and high-order components, and graphical tools. The popular Map-Reduce model (Dean and Ghemawat, 2008) also belongs to this class. *Composition in time* takes place when there are several tasks (or service operations) which have to be executed in the order of their temporal dependencies. Usually, there is a need for some external execution (workflow) engine which triggers activities and controls the order of execution. Many workflow systems are available for grids, including Kepler, Triana, Pegasus, and K-WfGrid (Gubała et al., 2006) systems. Challenges in scientific workflow applications and systems are outlined in Gil et al. (2007) and Zhao et al. (2009). On the other hand, scripting languages (Ousterhout, 1998) are useful for that purpose, since they provide constructs such as pipes and loops which allow the full expression of the complex control flow of the program.

In addition to composition in space and composition in time we should mention parallel and structured component composition. One approach is investigated in the CCA (Armstrong et al., 2006) model, taking into account such issues as data redistribution for MxN component connections (Bertrand et al., 2005). Parallel extensions to component models are introduced to the Corba Component Model (CCM) (Perez et al., 2003). Another type, which can be useful for more distributed and loosely-coupled scenarios, appears as *component collections*, as in the XCAT framework (Govindaraju et al., 2003) or the ProActive implementation of the Fractal component model (Baduel et al., 2006) with Grid Component Model (GCM) (Baude et al., 2009) extensions, including collective interfaces (Baude et al., 2007). The skeleton approach can be used, as in ASSIST (Danelutto and Aldinucci, 2006) and HOC-SA (Dünnweber and Gorlatch, 2009). 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 composition types, whereas, for example, pure service-oriented models do not allow (or do not directly support) composition in space. The examples of the XCAT and ICENI (Mayer et al., 2003) frameworks suggest that it is possible to combine both composition types in a single high-level model. Bouziane (Bouziane et al., 2008) suggests a graph-based notation which does not necessarily imply a simple solution. It is noteworthy that composition can also be applied to Web services, as it is in the Service Component Architecture (SCA) (Beisiegel et al., 2007).

The problem of obtaining access to remote computing resources is addressed in the following ways. In Globus and Unicore (Streit et al., 2005), virtualization is applied at the job processing level, whereas in Legion it reaches the higher software object level. In the case of service-oriented architectures, 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.

E-Science applications are often custom-developed and can evolve during their development and the lifecycle of the scientific experiment – thus the process of application deployment becomes a challenge. Grid middleware, such as Globus Toolkit, provides the low-level means of application installation on the execution host by the mechanism of *staging*. Another technology-enabling application deployment is virtualization (Sotomayor et al., 2008). A similar approach is offered by the *cloud computing* initiatives and the *Infrastructure-as-a-Service* model. Solutions such as the Amazon Elastic Compute Cloud (EC2) (Murty 2008) allow the deployment and running of virtual machine images on a configurable infrastructure. These solutions 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. Component technologies include the deployment process *directly into the programming model* and in the standards (Object Management Group, Inc., 2006b; Baude et al., 2009), since a component by definition is the basic unit of deployment.

As an alternative solution to the resource sharing problem H2O, a lightweight resource-sharing platform was proposed (Kurzyniec et al., 2003a). In H2O, resource providers only need to install an H2O kernel which serves as a basic container for deploying components, called pluglets; thus in H2O virtualization is applied at the container (H2O kernel) level. 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. As a lightweight deployment mechanism, H2O uses Java dynamic class loading features which allow the deployment and launching of any Java classes published remotely (and possibly packaged as JAR files) on HTTP or FTP servers. In addition to deploying Java classes, H2O provides a staging mechanism, which can be used to transfer arbitrary files, including native libraries or programs which are made available by the container to the pluglets. However, the use of native code in Java with JNI will sacrifice the portability of components and isolation between them, so this trade-off has to be taken into account. One of the advantages of H2O is that it uses RMIX (Kurzyniec et al., 2003b) as a multiprotocol communication library. Being similar in concept to other efficient implementations of RMI, such as Ibis (van Nieuwpoort et al., 2005), RMIX offers several benefits, including asynchronous and one-way calls as well as communication in peer-to-peer networks based on JXTA (Jurczyk et al., 2006).

The discussion of the features of the main programming models presented above allows us to draw some general conclusions about their advantages and disadvantages. 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. Although the MPI model (since version 2.0) supports dynamic creation of processes and communication establishment, it still does not provide high-level composition or deployment mechanisms. Distributed objects lack deployment and composition support in the model, so these important features need to be externally provided. The component model compares favorably with others, since it supports composition *and* deployment directly in the model.

The Common Component Architecture has unique features, which are of particular interest from the point of view of scientific applications. It was from the beginning designed to support scientific applications by introducing the Scientific Interface Definition Language (SIDL), which, together with the Babel system (Kohn et al., 2001), adds support for such datatypes as complex numbers, multidimensional arrays as well as multiple programming languages such as FORTRAN, C++ and Python. This makes CCA distinct from the Corba Component Model (CCM) (Object Management Group, Inc., 2006a) or Fractal (Bruneton et al., 2006). One of the advantages of a CCA specification is its simplicity: it is achieved by defining only a set of basic interfaces for interactions of components with the framework and for building applications, without making assumptions as to whether components are local or distributed. The CCA model is also dynamic: it allows the definition of component interfaces (ports) at runtime, which may be useful in dynamic composition and interaction scenarios.

We have to note that other component standards, such as CCM or Fractal (and its extended version, GCM), also have interesting features. CORBA was designed from the beginning to support distributed computing, so it standardizes, for example, Naming Services or deployment mechanisms. Fractal offers hierarchical composition of components and separates functional component interfaces from more generic controllers: these features have been exploited in GCM to support hierarchical deployment and autonomic controllers (Baude et al., 2009). However, we believe that the most important features of component models in general, such as communication using well-defined interfaces, third-party composition, and deployment support are common to all the component standards and implementations. Moreover, they can be used to provide interoperability between frameworks as we show in Section 3.6. We consider the choice of the CCA model as a practical decision, which helps focus and narrow down the research work while at the same time leveraging the benefits of the chosen solution.

Below we briefly synthesize how the related solutions address the key issues.

Facilitating high-level programming 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.

Facilitating deployment on shared resources Deployment on shared resources is handled by most grid-oriented frameworks, where descriptor-based solutions are the most popular – ADAGE, GEA, Virtual Nodes in ProActive and GCM (Coppola et al., 2005; Baude et al., 2009). In the case of Web services, deployment is not covered by specifications.

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

Communication adjusted to various levels of coupling Most of the frame works 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.

Adapted to the unreliable distributed environment 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.

Interoperable 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.

Secure Most of the frameworks support Grid Security Infrastructure as the most widely used standard. Shibboleth support is provided in some Web services based frameworks, including Globus WSRF.

The conclusion is that although the component model remains the most appropriate one for scientific applications on distributed e-infrastructures, none of the environments fully support all the features which have been identified as important (Section 1). This conclusion motivates us to work on new solutions, which is the subject of this paper.

3 Basic methods and tools

Having selected the component model in general, there is a need to focus on a concrete model and choose a base platform for constructing the environment. In this research the **CCA** as a component model and the **H2O** platform as a technology were selected. This decision introduces several benefits, some of which are immediate and result from the features of CCA and H2O, and some of which have to be elaborated upon and result in the higher layers of the proposed environment.

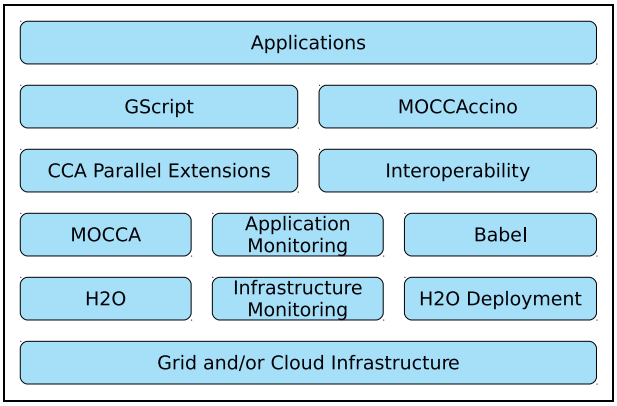

Figure 1. The layered architecture of the environment, from top: applications, high-level composition layer, parallel and interoperability extensions, base component frameworks, middleware technologies and low-level grid or cloud infrastructure.

Component models, and particularly CCA, offer APIs for component composition. To facilitate *high-level programming*, we propose to extend the composition mechanisms with a *high-level scripting language* or, alternatively, with a manager system based on the *architecture description language*. For *facilitating deployment* we exploit the dynamic deployment mechanisms offered by the H2O platform and we propose methods for creating a pool of H2O containers dynamically in existing grid or cloud infrastructures. Support for *diverse environments* is possible with 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 we propose extensions for multiple connections between CCA components. Optionally, CCA gives the possibility of using the Babel system for multilanguage support. To ensure that applications can be *adapted to the dynamic distributed environment*, the solution is to rely on the dynamic reconfiguration mechanisms of CCA and H2O, and to incorporate automatic management features into the high-level programming layers. To provide interoperability we propose to use Web services, but at the same time we develop a solution for integrating CCA components with GCM components, implemented in ProActive. Finally, regarding *security*, we have extended H2O to support the Grid Security Infrastructure (GSI) (Foster et al., 1998) and Shibboleth (Morgan et al., 2004) security solutions.

The concept of the environment can be presented as a layered architecture, outlined in Figure 1. On top, there is the scientific application which 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 (Malawski et al., 2008a) and the descriptor-based MOCCAccino (Malawski et al., 2006a)

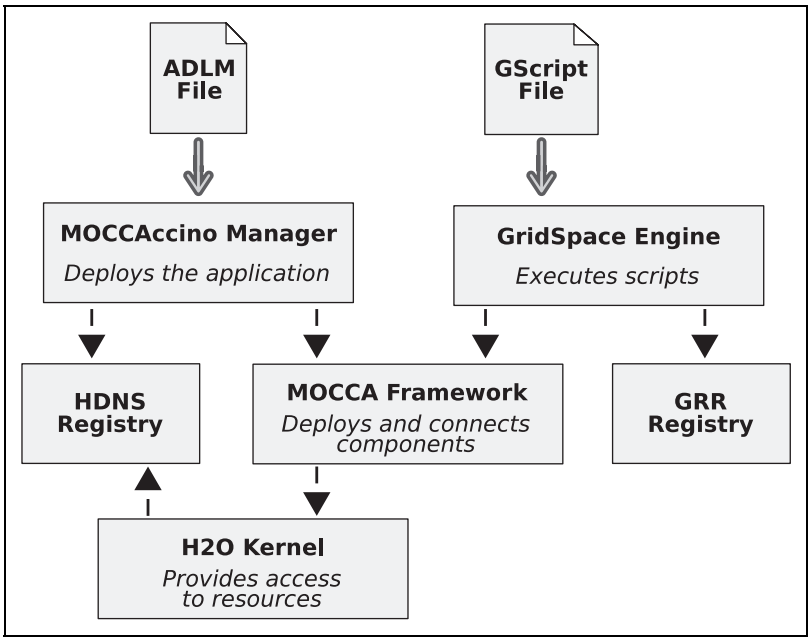

Figure 2. E-Science application composition with GridSpace and MOCCAccino as complementary scripting- and descriptor-based approaches.

system for composition based on the architecture description language. As discussed in Section 2, 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 development, experiments and steering, while MOCCAccino should be used for structured applications which require automated management.

The above-mentioned layers are built on top of base component frameworks. MOCCA (Malawski et al., 2005) 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, providing programming language interoperability. Below this layer there are basic middleware technologies which include H2O as a resource sharing platform and execution environment, and infrastructure monitoring providing system status information and techniques for deployment and management of the pool of resources. The lowest layer is the grid and cloud infrastructure which may include many different middleware types, as the role of higher layers is to hide them from the component model and the application itself.

3.1 Facilitating high-level programming

The chosen programming model should allow the composition of the application by third parties from smaller blocks (modules), and should express temporal dependencies between them, as well as direct connections. Combination of composition in time and in space is a crucial feature of the model, since both types of interactions are present in e-Science applications. 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.

In particular, 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 this, in order to program at a high level of abstraction and to hide the details of the underlying computing infrastructure, a high-level scripting layer and an Architecture Description Language-based layer is built. The support for both models is shown in Figure 2.

A **high-level scripting** layer is provided to enable application construction using an imperative language. By using a user-friendly API implemented in an object-oriented Ruby script, it is possible to compose the application at a high level of abstraction, while the underlying runtime system will be responsible for automatic component placement. Additionally, *the same* Ruby script (referred to here as **GScript**) is used to invoke operations on the created components, using control structures (loops, conditions, iterators, etc.), and 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, where the GridSpace engine is the core of the runtime system and the GRR registry stores the information about available components.

As an alternative approach, we offer an option to specify the application using a declarative language, namely an ADL. It enables hierarchical composition of component groups, where the actual number of components can be parametrized and dynamically managed. Such ADL-

based composition is realized in the **MOCCAccino** system, which uses the HDNS (Gorissen et al., 2005) registry for locating H2O kernels. MOCCAccino (Malawski et al., 2007a) defines an XML-based ADL for MOCCAccino (ADLM) description format, whose key features are parametrized groups of components which can recursively include other component groups and handle their connections. The MOCCAccino manager allows visualization of component groups, managing the internal representation of application structure, and deploying the component on the containers provided by the HDNS registry. MOCCAccino has been designed to be compatible with the CCA component standard. However, as new standards such as GCM emerge, we will consider adopting them as part of our future work.

We consider MOCCAcino and GScript as alternative and complementary approaches, but they are distinct models and it is as yet not possible to combine them together. We have considered two approaches to combining these models: either to generate GScript from the ADL descriptor or to construct the descriptors using the scripting API. Recently, our scripting approach has been more actively developed in the scope of the GridSpace virtual laboratory (Ciepiela et al., 2010) environment and its planned support for other application description languages, such as Multiscale Modeling Language (MML) (Falcone et al., 2010).

Our environment supports the composition types listed in Section 2 (low-level API, scripting languages, descriptor-based programming (ADL), and graphical tools), apart from skeletons and high-order components. However, these constructs can, to some extent, be modeled either by constructing parametrized scripts or ADL descriptors. Moreover, as one of our goals was to support flexibility in programming and experimentation, we found the high-level scripting approach the most promising.

3.2 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. The deployment should be dynamic, allowing adaptive application behavior, by providing the capabilities for deployment, undeployment and redeployment of code at runtime.

In the component model, the concept of a component container and the deployment process are reflected directly. 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. The selection of H2O as the component container solves the basic deployment problems, since the H2O kernel is a fully-fledged application server with remote and dynamic deployment capabilities.

By selecting a component model, the problem of application deployment can be reduced to the problem of deployment of components into a container. Therefore, assuming that a pool of H2O kernels is available, the underlying grid infrastructure is *virtualized* as a pool of component containers. Using cloud terminology, the virtualization layer of H2O and MOCCA can be considered as a Platform-as-a-Service (PaaS) layer positioned above the Infrastructure-as-a-Service stack.

Unfortunately, in current production infrastructures such as EGEE/EGI, it cannot be assumed that a pool of containers is automatically available, so 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 Malawski et al. (2006b). The idea is to use the concept of *pilot jobs*, known from e.g. Condor (Thain et al., 2005), to spawn the required number of H2O kernels as grid jobs using available middleware. Additionally, to support communication between components running in private networks of multiple clusters, it is possible to use the JXTA P2P overlay network which was integrated with our system (Jurczyk et al., 2006). This solves the problem of connecting machines which cannot communicate directly because of NAT or firewall restrictions, which is often the case.

The same mechanism of dynamic provisioning of component containers is even simpler when using a cloud platform, such as Amazon EC2. We have prepared the Amazon Machine Image (AMI) with an H2O kernel installed and preconfigured to automatically start at system boot time and to listen on the public interface. Using a simple API it is thus possible to dynamically, on demand, add component containers to the resource pool. It is noteworthy that adding the support for EC2 was a straightforward task, which confirms that the component-based approach fits well with the cloud infrastructure. Moreover, as the performance experiments with the EC2 cloud indicate, the deployment times of virtual machines are on average less than 2 minutes (Ostermann et al., 2010), which gives some flexibility in terms of adjusting the size of the resource pool.

The deployment process on grids and clouds as described above is schematically depicted in Figure 3. First, the user creates a pool of H2O kernels using the API for grid or cloud infrastructure. Once the kernels are running, the component application can be deployed into the kernels using standard CCA API or tools.

One observation is noteworthy with respect to the resource-sharing model. Since our solution is based on the H2O lightweight platform, it is possible to use resources either directly accessible via H2O, or to harness additional resources from grid or cloud infrastructures by deploying H2O on top of them. This approach isolates the component application from the low-level mechanism of resource provision and for that reason it was possible to add support for new infrastructures, such as Amazon EC2, without significant effort. This means that by creating a virtualization layer of component containers it is possible to hide the differences between grid and cloud from the application perspective.

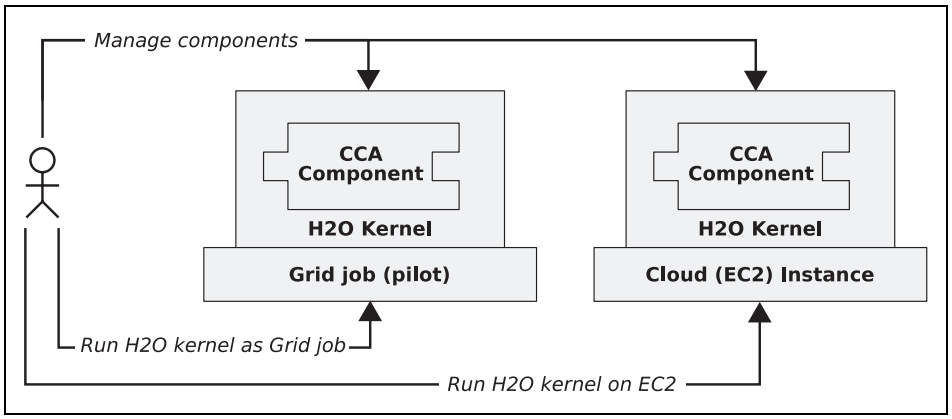

Figure 3. Deployment of component containers (H2O kernels) as pilot jobs on grid nodes or as virtual machines on the cloud. Once the pool of containers is available, the underlying infrastructure is hidden to the component framework.

3.3 Scalability to diverse environments

Scientific applications often involve various computation models simulated with specific, optimized environments. To support this requirement the proposed framework should be scalable to run on machines ranging from single PCs or laptops, through HPC clusters to multiple grid and cloud sites. In other words, the environment should guarantee that the underlying infrastructure does not determine the programming model. The concept of a lightweight container and the mechanism of component composition allow the creation of applications in a dynamic, pluggable way, thus fitting heterogeneous environments.

In order to achieve the goal of scalability to diverse computing bases, the environment should be based on two principles: a lightweight platform and mechanisms for pluggable and reconfigurable extensions. H2O can serve as a lightweight platform, since it only requires a Java 1.4 or newer virtual machine (which provides portability), runs out-of-the-box from a 20 MB packaged installation, takes around 1 s to start up on a 2 GHz PC, and has a small memory footprint (approximately 25 MB). 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 EGI or Amazon EC2. 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.

3.4 Communication and 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 the application modules to these physical constraints. The communication layer should also support collective or parallel connections between application modules.

In the component model, 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 in both local and distributed configurations, while the protocol layer is managed by the framework. The component models also allow for parallel or group connections and communications.

H2O offers a multiprotocol communication library called RMIX for 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.

One of the advantages of the CCA component model is that it supports components developed in multiple programming languages using Babel (Kohn et al., 2001). Babel was extended to provide remote method invocation (Babel-RMI) (Kumfert et al., 2007). Our early experiments with combining Babel-RMI with RMIX (Malawski et al., 2006c) show the advantages of such a multilanguage and multiprotocol solution.

As 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 a MultiBuilder extension (Malawski et al., 2006b) which allows the creation, connection and invocation of collections of components with parametrized number of instances. The parallel connections between component groups are handled by the MOCCAccino ADL and manager system.

3.5 Adaptability to grid and cloud environments

As e-Science applications tend to use large-scale computation components, use of the vast, powerful computing environments of the grid is a natural requirement.

However, as such environments 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. 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.

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 to (or removed from) the resource pool. These are 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.

In order to adapt GScript to a dynamic grid infrastructure, we introduced the GridObject abstraction (Malawski et al., 2010) into our high-level scripting model. GridObjects are used in the script to represent components, Web services, or other executable entities such as programs submitted to the grid as computing jobs. The main feature of GridObjects is that the programmer can specify only their class in the script, and the environment will select or create the appropriate instance at runtime. Subsequently, it is possible to invoke operations on such GridObjects in a similar way as on ordinary objects in the script. To deal with the problem of component placement or optimal instance selection, GridSpace includes the optimizer module, which chooses the optimal resource based on the monitoring information and simple heuristics (Malawski et al., 2008b).

3.6 Interoperability

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, in principle, possible. However, the fact that RMIX does not support WSDL becomes an issue. Therefore we consider using an 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 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 to allow interframework interoperability. We developed a solution based on the adapter concept which enables both types of interoperability between CCA and GCM (Malawski et al., 2007b).

3.7 Security

Security is an important requirement for a system which allows the deployment and execution of custom application code on remote and shared resources, including proper authentication, authorization, and transport security. For large-scale systems with multiple computing nodes in multiple administrative domains, additional requirements are for Single Sign-On (SSO) and credential delegation, i.e. allowing a process running on a remote node to access resources on another one on behalf of a user.

The component model helps achieve separation of concerns by introducing the concept of a container. The security aspects such as authentication, authorization, and transport security can be managed and configured by the framework. The container can provide sandboxing to protect the code running on shared computing resources from interfering with other code.

The H2O kernel is a component container which provides pluggable authentication modules and flexible authorization policies. Transport security is assured by the RMIX communication library which supports SSL, while sandboxing is provided by the H2O kernel using Java security features. However, it is noteworthy that due to limitations of the Java platform (as described in Section 2), the trade-off between performance of native code and security has to be taken into account.

The first extension which we introduced into the environment was the integration of H2O with Grid Security Infrastructure (GSI) (Foster et al., 1998). This solution uses X.509 certificates with proxy extensions, which provide SSO and credential delegation, as well as compatibility with most production grid infrastructures such as EGI. As a result, access to the MOCCA framework can be granted only to such clients who can provide a valid proxy certificate for a registered user (Dyrda et al., 2009).

Shibboleth (Morgan et al., 2004) is a federated Web Single Sign-On framework based on SAML (Security Assertion Markup Language). SSO is achieved by letting users use their home organization logins and passwords to access remote resources. Shibboleth features attribute-based access control, and by mutual agreement between participating institutions it allows the decentralized building of virtual organizations. We implemented the Shibboleth-based authenticator in the H2O kernel, in which a Shibboleth handle is used as a credential and an external

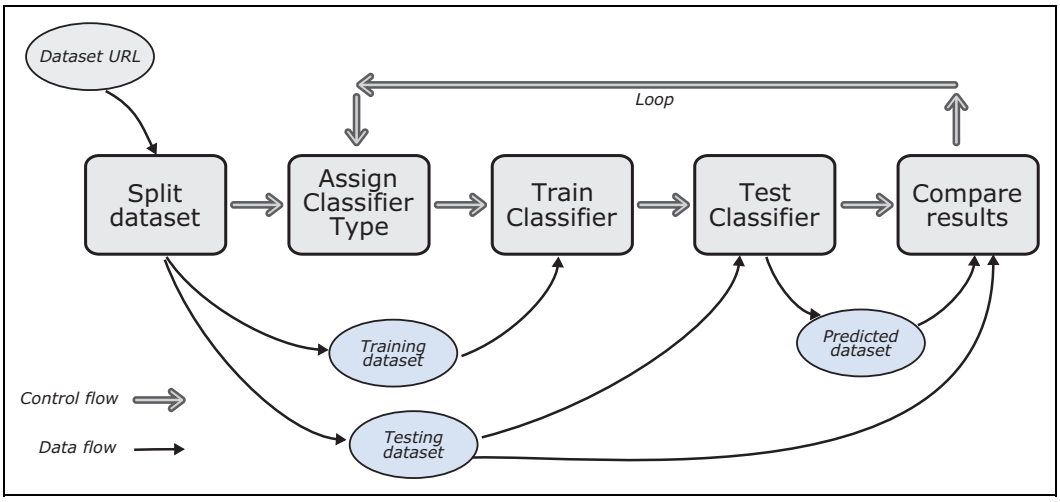

Figure 4. The data and control flow for the sample script demonstrating the use of the Weka Data Mining application which uses MOCCA components.

policy decision point (PDP) is used for authorization. The advantage of Shibboleth over GSI is that the users are not required to have their certificates, but the problem is a lack of proper management of security handles for long-running computations. The Shibboleth authenticator enabled the integration of MOCCA with the virtual organization infrastructure which controls access to the ViroLab virtual laboratory (Meizner et al., 2009).

4 Case studies and experiments

In this section we present the results of the experiments which demonstrate the applicability of the component approach for the sample applications. The first experiment presents the usage of the scripting approach on the example of a data mining application. The second experiment involves an application which simulates the formation of gold clusters using the simulated annealing method. Finally, we show the results of the synthetic benchmark prepared to measure the overhead of the component framework.

4.1 Weka experiments in ViroLab

The ViroLab virtual laboratory (Bubak et al., 2009) 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.

MOCCA is one of the supported middleware technologies, and the GridSpace scripting engine, as described in Section 3.1, 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 (Witten and Frank, 2005) library wrapped in components.

Key functionality elements of Weka, such as classifiers, association rules, clustering algorithms, and filters, were wrapped as components to allow for more flexible creation of various experiments. 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 a *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 4 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 as shown in Figure 5. 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*.

The Weka experiment demonstrates how our environment can be used to support a pure *composition in time* scenario. Components are not connected and do not interact directly, instead they are controlled by an execution engine which invokes operations on their ports. The experiment demonstrates also how the GridObject abstraction introduced in GridSpace allows the *orchestration* of components together with Web services in a transparent way: the actual implementation of *WekaURLGem* is a stateless Web service, while classifiers are MOCCA components.

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).

```
Classifiers = [
       'weka.classifiers.rules.Prism',
       'weka.classifiers.functions.Logistic',,
       'weka.classifiers.trees.J48',
       'weka.classifiers.lazy.KStar'
      ]

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

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

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

i = 0
10.times do
  classifier.assignClassifier(Classifiers[i])
  learning_time = classifier.trainURLdata(
       splitURLData.trainingURLdata, '', 'class')

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

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

  result = classificationPercetnage.to_f * 100.to_f

  wekaURLgem.deleteURLdata(classifiedData)

  i = i + 1
end
```

Figure 5. Data mining application script. **GObj.create()** deploys the component which can subsequently be used by invoking operations directly from the script.

4.2 Application: Gold cluster formation

The goal of the second application has been to demonstrate how *composition in time* can be combined with *composition in space* in a more complex real application scenario.

The formation of clusters of gold atoms is an important process in nanotechnology (Wilson and Johnston, 2000). The goal is is to apply a simulated annealing method to minimize the energy of the molecules, given the molecule size and the potential. The application is compute intensive, and it requires not only minimization of the energy, but it involves a larger loop, in which the actual minimization method is optimized by tuning parameters such as cooling function or initial configurations. The component-based application for simulating the formation of gold clusters has evolved over time. Below, we describe a version where the energy minimization is additionally the subject of an automatic tuning of the application parameters (see Figure 6).

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 is designed to support a single *Molecule* provider; the *Gather* component handles multiple connected components and passes their results to the *Storeroom*. This enables building a hierarchical tree of gather components, which may be required when deploying the application on a large number of nodes.

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

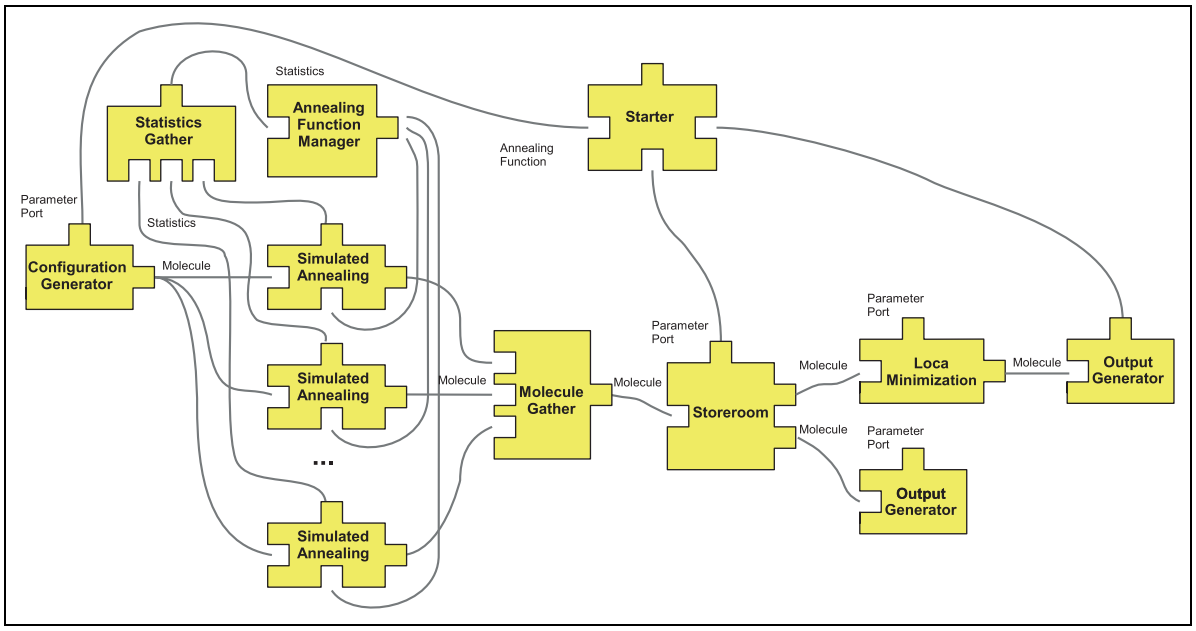

Figure 6. Configuration of gold cluster application enabling parameter tuning in order to optimize the energy minimization process.

improve the results using the L-BFGS method (using the JAT (Berhold and Takada, 2002) library). For interactive visualization, a prototype version of the *Output Generator* component was developed using the Jmol (Herráez, 2006) visualization library (not shown in the diagram).

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 manage the queue of data items to be processed. The gather functionality has been abstracted so that it can be reused in other applications. The components in this scenario were deployed using Ruby script and MultiBuilder mechanism (see Section 3.4), while the number of components is parametrized in the script.

By following a similar approach to the one described in Malawski et al. (2006b), it was possible to deploy the simulated annealing application on the French Grid'5000 testbed. The application was successfully deployed on three clusters located at Sophia-Antipolis, Bordeaux, and Orsay, and the computing times and throughput for the molecules of 20 atoms were measured. Figure 7 presents results of one of the experiments, showing the throughput in molecules per minute versus the number of cores used. Although the results indicate that it is possible to achieve a good speedup with our framework, the main advantage of the component approach is the flexibility of application composition and facilitated adaptation to new environments, such as Grid'5000.

The gold cluster simulation has demonstrated how the framework can be used for composition in space, i.e. the components interact directly as in a choreography scenario where the components communicate by involving operations on connected ports. Moreover, this can be combined

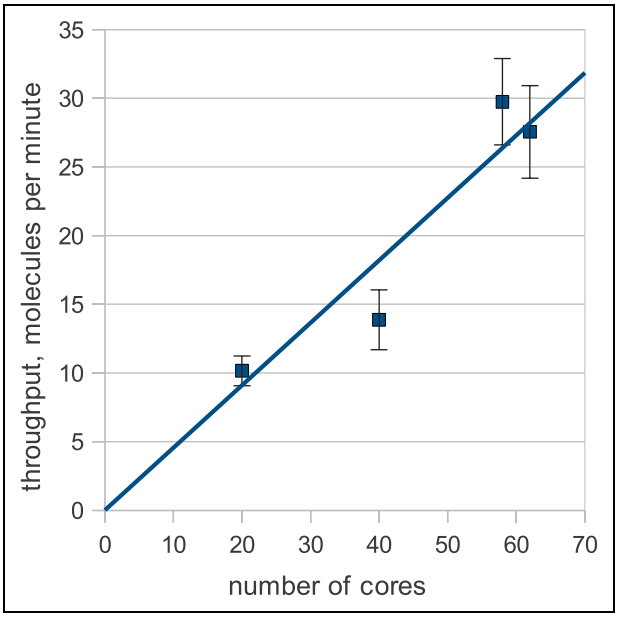

Figure 7. Throughput of the gold cluster application run on 3 clusters of Grid'5000 (Sophia, Orsay, Bordeaux). The line shows a linear fit $y=(0.45\pm0.12)x$, $R^2=0.89$; the error bars represent the standard error of the fit.

with composition in time, as it is possible to invoke operations on components directly from the script. This can be used, for example, for interactive steering of computations. Another benefit of the component approach is the flexibility of constructing different application scenarios on different resources while reusing the basic components. The same set of components has been used to deploy the application locally with no parallelism, to run them on a local cluster, and also on a distributed grid infrastructure.

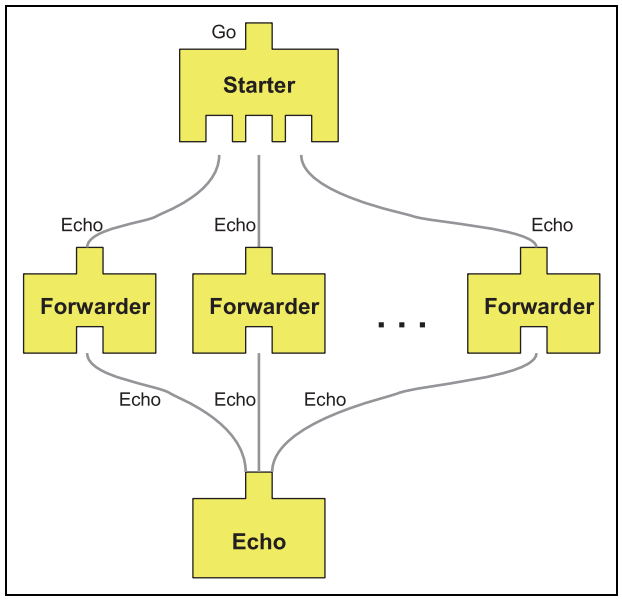

Figure 8. Configuration of components in the benchmark application. The number of *Forwarder* components in the collection is parametrized.

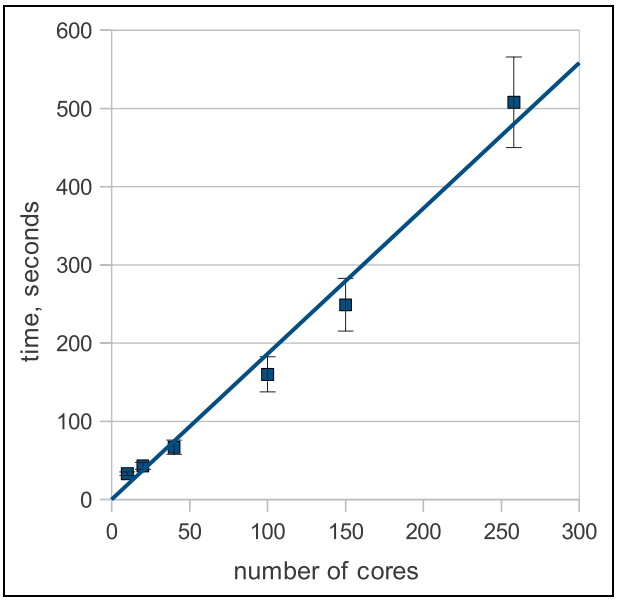

Figure 9. Total execution time of the benchmark application (version 1) on 258 cores, 6 clusters. The line shows a linear fit $y=(1.86\pm0.12)x$, $R^2=0.98$; the error bars represent the standard error of the fit.

4.3 Scalability experiments on Grid'5000

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

The structure of the application is shown in Figure 8. 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 several-byte string message. The components were created using the *MultiBuilder* mechanism introduced in Section 3.4. The goal of ***version 1*** of the benchmark was to measure execution times where all the stages were performed *sequentially*, so no parallelism was exploited.

First, the application was run on a pool of 114 H2O kernels running on 114 nodes of 6 clusters, totaling 258 cores; the number of *Forwarder* components in the collection was equal to the number of cores. The total run time (from client startup to the end of cleanup) versus the number of cores is shown in Figure 9 It can be seen that the growth of computing time is nearly linear with respect to the number of components (cores). This can be explained by the fact that all operations (deployment, connection, invocation, and destruction) were invoked *sequentially*. As in other multicluster experiments reported here, the linear fit is a simplification since increasing the number of CPU cores required access to more computing clusters (up to 6 for 258 cores). From the fit coefficient we can estimate that the average processing time per component was less than 2 seconds, which is comparable to the time for running the above application on a single node. The conclusion is that creation of a large number of connections between components using the *MultiBuilder* mechanism does not introduce additional overhead. This means that the environment preserves scalability when handling collections of components sequentially.

Table 1. Sample results for the duration (in seconds) of application stages (version 1) for two numbers of computing cores (n).

clusters	n	creation	connection	computing	destroy	total
7	260	207	25	219	99	551
6	240	90	20	171	103	384

The goal of the second experiment was to measure the time of each stage of the benchmark application. Preliminary measurements were performed on the same sequential version of the application. To illustrate the high variability of results when executing the application on Grid'5000, the results of the two sample runs are shown in Table 1. There are many factors which influence the performance in a heterogeneous grid environment; for example, for deploying 240 components we needed machines from 6 clusters, while 260 components required allocating them on one more cluster. Despite that observed variability, it is possible to assess which stages have the highest time cost. 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 relatively long, since it involves opening new sessions to H2O kernels and instantiating a new component, including class loading. The reason behind the lengthy

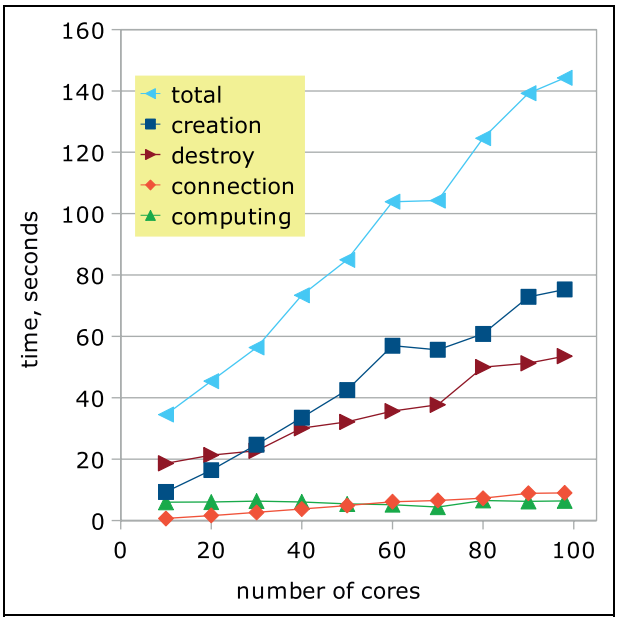

Figure 10. Detailed execution times of the benchmark application (version 2) on 100 cores of 6 clusters.

computation time stems from the implementation of the 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 *Forwarder* instance, which explains the delay and overall time.

The goal of the ***version 2*** of the benchmark was to measure the performance of parallel invocation of operations on the collection of components. The implementation of concurrent invocation in the *Starter* component is based on the *cached thread 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 Figure 10. This time, the computation time is reduced to approximately 5% of total run time, while for the sequential version it was nearly 50%. As expected, the asynchronous execution considerably improves the application performance, but still we can observe overhead induced by the initial opening of the connections.

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 perform a series of invocations of the echo operation after obtaining a reference to the port (***version 3***). The time of the first invocation (labeled computing1) was measured separately from the average time of the 10 subsequent invocations (labeled computing). The results are presented in Figure 11(a), with the computing time (enlarged scale) shown in Figure 11(b). It can be seen that the computation time for 10 components (cores) is 0.2 s 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 observed that the invocation (computation) time grows 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 the 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 benchmarks described above it was possible to deploy and run the test application on 600 and 800 cores of 8 clusters each. The results shown in Table 2 are in agreement with the 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. The results shown here were obtained for the scenario where all the stages except computing were performed sequentially. However, the average times required to deploy and destroy a single component are in the order of 0.5 s.

The results of the large-scale deployment experiments are 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. The times of various steps of the application lifecycle were measured and the observed behavior was explained.

Finally, we can conclude that the component-based approach does not introduce significant overhead and the environment retains scalability even for large-scale deployments which are typical for grids and clouds. These results are consistent with those yielded by tests of other Java-based frameworks such as ProActive or Satin (Van Nieuwpoort et al., 2010).

5 Analysis

By combining matching concepts of CCA and H2O, we have shown that it is possible to develop a programming environment which satisfies the requirements of e-Science applications and is capable of exploiting the capabilities of grid and cloud infrastructures. 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. The applications described in this paper have been successfully developed and run using the scripting approach.

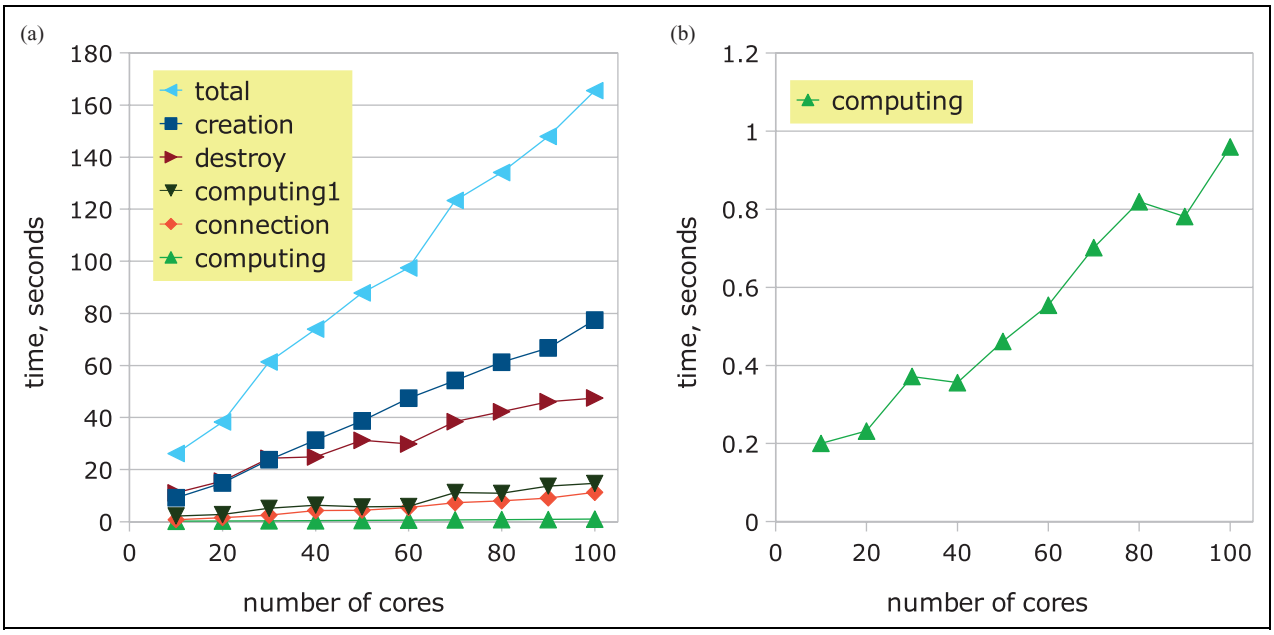

Figure 11. Detailed measurements of the benchmark application (version 3) on 100 cores of 4 clusters. (a) Execution times of the stages of the benchmark application. (b) Average execution time of the computing stage (enlarged).

Table 2. The duration of subsequent stages of application deployment on up to 800 cores of 8 clusters. The number of cores is denoted as *n* and the execution time is given in seconds.

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

Facilitating deployment on shared resources of (possibly custom) component code is possible thanks to H2O dynamic deployment mechanisms. When utilizing existing grid infrastructures, such as EGI, a dynamically managed pool of component containers can be created; moreover on clouds such as EC2 it can be achieved with even less effort. This was demonstrated by running the gold cluster simulation and test benchmarks on these infrastructures. Moreover, deployment times were measured and explained.

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 and cloud infrastructures.

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. This feature was assessed with the case studies described in Section 4, as well as in our earlier experiments (Malawski et al., 2005, 2006b; Jurczyk et al., 2006).

Adapted to the unreliable distributed environment This is achieved by combining dynamic capabilities of the CCA model and the H2O platform, which together enable dynamic deployment and composition. Moreover, the GridObject abstraction introduced in the GridSpace environment allows applications to be programmed using high-level scripts which are independent from the underlying runtime infrastructure.

Interoperability with other component models was demonstrated in the example of the Grid Component Model and the ProActive framework, with the connection of our gold cluster simulation application with additional components developed in ProActive, described in detail in Malawski et al. (2007b).

Security was assured by leveraging built-in mechanisms of the H2O container, and developing and integrating extensions to support the most widely used authentication systems, GSI and Shibboleth. This was particularly important as MOCCA was one of the middleware technologies supported by the ViroLab virtual laboratory which was based on GridSpace (Meizner et al., 2009).

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 by offering high-level application composition based on a scripting approach which is technology-neutral. Moreover, the interoperability experience with GCM and ProActive 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 experience with the development of our component environment and the example applications allows us to draw some general conclusions about component models for large-scale scientific applications.

We can conclude that the general advantages of component models, such as clear definition of interfaces, support for third-party composition and definition of deployment units have proven to be really useful and facilitated the development and execution of applications. For example, it was possible to substitute the components in the gold cluster simulation application or the Weka experiments for other implementations in a flexible way. Moreover, changing the application configuration by adding new components, or adjusting it to different deployment configurations, was easy as well. This is particularly important when the application is under continuous development and there is a need to migrate between a laptop and the local cluster of a grid testbed. This approach facilitates rapid prototyping of applications and the execution of many test runs which is a characteristic of e-Science. The component model enforces or encourages some good software engineering practices, such as modularity and extensibility, so these productivity aspects also have to be taken into account. For an additional example of the benefits of using component- or object-based approaches for more tightly coupled parallel applications, we refer to the research done within the scope of the ProActive framework (Baduel et al., 2005; Caromel and Henrio, 2005; Parlavantzas et al., 2007).

Regarding using H2O and Java more generally as the virtualization layer for our framework, we can see the obvious advantages of this approach, namely platform independence and dynamic deployment capabilities. Without them it would be virtually impossible to run our experiments on such a wide range of resources. Regarding the trade-off between portability and performance, we believe that given the inherent complexity of distributed infrastructures and abundance of compute resources available to scientists from various sources, the more significant problem is how to effectively harness the compute power that is available. Providing a virtualization layer (in this case a pool of component containers) simplifies the problem to large extent.

We can also contribute to the discussion comparing grid and cloud computing models. As noted in Section 3.2, it is possible to abstract the underlying resources by creating a virtualization layer in the form of pool of component containers. However, we observed that the cloud computing model, where the additional machines can be acquired on demand is more convenient and better matches the component paradigm than the grid model, where resources are accessible using job submission queueing systems. It is also possible to treat the cloud infrastructure directly as a large-scale distributed container, and the virtual machines as components which can be deployed thereon. This allows us to overcome some shortcomings of using Java as virtualization layer, but it introduces other challenges related to cloud computing, such as security, interoperability between clouds and virtual machine image management (Malawski et al., 2011).

Regarding communication models in distributed systems, we have observed also some limitations of our solution. 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). Our 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 at the level of the base component model, while the higher-level tools for component composition proposed in this paper remain valid and usable. As a part of our current and future work, we have begun adding support for communication using message queues such as Amazon SQS (Murty, 2008), which can be useful for reducing coupling between interacting components.

When comparing our solution to other component-based approaches, such as GCM, one can observe that there are many similarities which come from the general concepts of component models. The similarities are confirmed by the results of our efforts to provide interoperability between these component models and frameworks (Section 3.6). There are also some features which distinguish our solution. In our opinion the most valuable concept is the high-level scripting approach that we introduced in GScript and GridSpace (Section 3.1), and we can observe that similar high-level scripting approaches have recently gained importance also for petascale systems (Wilde et al., 2009).

On the other hand we observe that the problems related to deployment of applications on shared resources are becoming addressed by Web service and cloud computing activities. One of the examples is OSGi, which defines how to package and deploy software bundles, also in remote containers (Rellermeyer et al., 2007). On the other hand cloud computing IaaS infrastructures such as Amazon EC2 allow deployment of virtual machines on demand using an API very similar to the builder interfaces known from, for example, the CCA component model. By offering a different level of virtualization, they solve some of the issues we encountered with Java-based solutions, related to isolation and handling native code, but they introduce other problems characteristic of clouds.

One interesting observation comes from comparing our scripting-based solution to other workflow systems which can be used for grid and cloud computing, such as Pegasus (Deelman, 2010). The advantage of the workflow model is that by limiting the application structure (to directed acyclic graphs, for example) the problems of application

composition, planning and scheduling are simplified. However, it is not always possible or convenient to express a given application in the form of a workflow. On the other hand, our approach to composition using Ruby-based scripting offers greater flexibility, and the possibility of combining the composition script with all the programming language constructs and libraries available to the language. This on the one hand makes the problem of scheduling more complex due to implicit dependencies between operations invoked on GridObjects, but on the other hand it allows easy prototyping of applications and the addition of, for example, Web-based interfaces, which proved to be very useful for applications in the ViroLab Virtual Laboratory (Bubak et al., 2009).

Finally, we should emphasize how our solution is different from similar Java-based frameworks for distributed and grid computing.

- The main difference between H2O and ProActive is that ProActive assumes an underlying programming model based on active objects and restricts the deployment capabilities to such objects, while H2O does not have such constraints. For that reason it was convenient to build a CCA-compliant framework on top of H2O and later to interface with MOCCA from the GridSpace scripting environment. Moreover, we observed that the deployment descriptor mechanism in ProActive is quite complex and not easy to use, while our high-level scripting approach was from the beginning designed to insulate users from low-level deployment details. Additionally, we have added support for two important security solutions, namely GSI and Shibboleth, which were not available in ProActive.
- The main advantage of H2O with respect to IBIS (van Nieuwpoort et al., 2005) is the lightweight deployment mechanism and the container concept which fits the CCA component model very well. At the higher level, the developers of IBIS focused on supporting more specific programming paradigms, such as divide-and-conquer in Satin (Van Nieuwpoort et al., 2010), while our work was devoted mainly toward the high-level scripting approach and flexible combination of composition in space and in time.
- When comparing with higher-order components or skeleton frameworks such as HOC (Dünnweber and Gorlatch, 2009) or ASSIST (Danelutto and Aldinucci, 2006), or Map-Reduce implementations such as Hadoop, we find them valuable when dealing with more specific types of problems, e.g. Map-Reduce can be tailored for processing large-scale genomic data in the cloud, as in CloudBurst (Schatz, 2009). Conversely, we pursued mainly a scripting approach as a flexible way to support exploratory programming and more flexible experimenting.
- Regarding Web service based solutions, such as XCAT (Krishnan and Gannon, 2004) or SCA (Beisiegel et al., 2007), we acknowledge that the advantages of H2O are lightweight deployment mechanisms and more efficient communication, which are better suited for scientific applications.

As discussed in Section 2, many of the desired features are present in existing frameworks, but we believe that the combination of them which is offered by our environment constitutes an interesting contribution and may become useful for scientific applications.

6 Summary and future work

In this paper we analyzed the problem of programming complex scientific applications on grid and cloud infrastructures. 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 underlying distributed infrastructure and enhanced with a set of tools facilitating the programming of applications at 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 and security. Moreover, this approach facilitates rapid prototyping of applications and executing many experiment runs, which is typical for e-Science.

The main conclusion is that choosing a component model such as CCA and a lightweight resource sharing platform such as H2O is an appropriate solution for scientific applications on grid and cloud infrastructures.

The concepts and methods devised in this paper are of a general nature and can thus outlive specific technologies and implementations. Experience gained from experiments on constructing applications from components and providing higher-level tools and abstractions will be useful for both distributed computing technologies: grids and clouds. 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, IaaS cloud providers, through gaming consoles such as the PlayStation, to mobile devices. Specifically, adding new cloud providers to the resource pool would not change the way in which an application is constructed and deployed. This conclusion is in agreement with the one stated in Schwiegelshohn et al. (2010) that grid and cloud computing approaches can complement and benefit from each other.

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, which constitute alternative solutions to the component composition problem. The development of such models and tools remains a highly relevant research challenge.

A list of future research directions which were identified includes systematic development of supporting algorithms, for example for deployment planning; higher-level programming support using semantic Web concepts; development of a more integrated environment to make it more usable; development of a formal model which would enable reasoning about the properties of the environment and the application, etc.; and better support for wrapping legacy applications as components to enable the environment to be practically applicable in more real-life applications.

Acknowledgments

The authors would like to express their gratitude to multiple collaborating colleagues from CoreGRID and ViroLab projects. Special thanks go to Dawid Kurzyniec, Eryk Ciepiela, Joanna Kocot, Tomasz Bartynski, Daniel Harezlak, Michal Placek, Tomasz Jadczyk, Michal Dyrda, Jan Meizner and Piotr Nowakowski.

Funding

Maciej Malawski acknowledges the support by a grant ''Improvement of didactic potential of computer science specialization at AGH'' (grant number UDA-POKL.04.01.01-00-367/08-00). This work was partially supported by the EU Project *VPH-Share: Virtual Physiological Human: Sharing for Healthcare – A Research Environment* (contract number 269978). Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, (see https:// www.grid5000.fr). Access to the Amazon EC2 cloud was supported by an AWS in Education grant.

Conflict of interest

None declared.

References

Altunay M, Avery P, Blackburn K, Bockelman B, Ernst M, Fraser D et al. (2011) A science driven production cyberinfrastructure – the Open Science Grid. *Journal of Grid Computing* 9: 201–218.

Armstrong R et al. (2006) The CCA component model for high-performance scientific computing. *Concurrency and Computation: Practice and Experience* 18: 215–229.

Baduel L, Baude F and Caromel D (2005) Object-Oriented SPMD. In *IEEE International Symposium on Cluster Computing and the Grid*, Vol. 2, pp. 824–831.

Baduel L et al. (2006) Programming, deploying, composing, for the grid. In: Cunha JC and Rana OF (eds) *Grid Computing: Software Environments and Tools*. Heidelberg: Springer.

Baude F, Caromel D, Dalmasso C, Danelutto M, Getov V, Henrio L et al. (2009) GCM: a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications* 64: 5–24.

Baude F, Caromel D, Henrio L and Morel M (2007) Collective interfaces for distributed components. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)* 14–17 May 2007, Rio de Janeiro, Brazil. IEEE Computer Society, pp. 599–610.

Beckman PH (2005) Building the TeraGrid. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 363: 1715–1728.

Beisiegel M et al. (2007) *SCA Service Component Architecture Assembly Model Specification, Version 1.0*. http://osoa.org/display/Main/Service+Component+Architecture+Home

Berthold T and Takada N (2002) *Java Astrodynamics Toolkit (JAT)*. http://jat.sourceforge.net/

Bertrand F, Bramley R, Sussman A, Bernholdt DE, Kohl JA, Larson JW et al. (2005) Data redistribution and remote method invocation in parallel component architectures. *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings*, 4–8 April 2005, Denver, CO. IEEE Computer Society.

Bishop P and Warren N (2003) *JavaSpaces in Practice*. Boston, MA: Addison-Wesley.

Bolze R, Cappello F, Caron E, Daydé M, Desprez F, Jeannot E et al. (2006) Grid'5000: A large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications* 20: 481–494.

Bouziane HL, Pérez C and Priol T (2008) A software component model with spatial and temporal compositions for grid infrastructures. In: Luque E, Margalef T and Benitez D (eds) *Proceedings Euro-Par 2008 – Parallel Processing, 14th International Euro-Par Conference*, Las Palmas de Gran Canaria, Spain, 26–29 August 2008 (Lecture Notes in Computer Science, vol. 5168). Berlin: Springer, pp. 698–708.

Bruneton E et al. (2006) The fractal component model and its support in Java. *Software: Practice and Experience* 36: 1257–1284.

Bubak M, Gubała T, Kapałka M, Malawski M and Rycerz K (2005) Workflow composer and service registry for grid applications. *Future Generation Computer Systems* 21: 79–86.

Bubak M, Malawski M, Gubala T, Kasztelnik M, Nowakowski P and Harezlak D (2009) *Virtual Laboratory for Collaborative Applications*. IGI Global, Chapter 27: 531–551.

Bubak M, Malawski M and Zajac K (2003) Architecture of the grid for interactive applications. In: Sloot PMA et al. (eds) *Proceedings of Computational Science – ICCS 2003, International Conference*, Melbourne, Australia and St. Petersburg, Russia, June 2003 (*Lecture Notes in Computer Science,* vol. 2657). Berlin: Springer, pp. 207–213.

Caromel D and Henrio L (2005) *A Theory of Distributed Objects*. Berlin: Springer.

Ciepiela E, Harezlak D, Kocot J, Bartynski T, Kasztelnik M, Nowakowski P et al. (2010) Exploratory programming in the Virtual Laboratory *Proceedings of the International*

Multiconference on Computer Science and Information Technology, Wisla, Poland. IEEE Press, pp. 621–628.

Coppola M, Danelutto M, Lacour S, Pérez C, Priol T, Tonellotto N et al. (2005) Towards a common deployment model for grid systems. In: Gorlatch S and Danelutto M (eds) *Proceedings of the Integrated Research in Grid Computing Workshop*. Pisa, Italy: Università di Pisa, Dipartimento di Informatica, TR-05-22: 31–40.

Danelutto M and Aldinucci M (2006) Algorithmic skeletons meeting grids. *Parallel Computing* 32: 449–462.

Dean J and Ghemawat S (2008) MapReduce: Simplified data processing on large clusters. *Communication of the ACM* 51: 107–113.

Deelman E (2010) Grids and clouds: Making workflow applications work in heterogeneous distributed environments. *International Journal of High Performance Computing Applications* 24: 284–298.

Dünnweber J and Gorlatch S (2009) Higher-Order Components for Grid Programming – Making Grids More Usable. Berlin: Springer.

Dyrda M, Malawski M, Bubak M and Naqvi S (2009) Providing security for MOCCA component environment. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009*, Rome, Italy, 23–29 May 2009. IEEE Press, pp. 1–7.

Falcone JL, Chopard B and Hoekstra AG (2010) MML: Towards a Multiscale Modeling Language. *Procedia CS* 1: 819–826.

Foster I and Kesselman C (2006) Scaling System-Level Science: Scientific exploration and IT implications. *Computer* 39: 31–39.

Foster IT (2006) Globus Toolkit Version 4: Software for service-oriented systems. *Journal of Computer Science Technology* 21: 513–520.

Foster IT, Kesselman C, Tsudik G and Tuecke S (1998) A security architecture for computational grids. In *ACM Conference on Computer and Communications Security*, pp. 83–92.

Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM et al. (2004) Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, pp. 97–104.

Gentzsch W, Girou D, Kennedy A, Lederer H, Reetz J, Riedel M et al. (2011) DEISA – Distributed European Infrastructure for Supercomputing Applications. *Journal of Grid Computing* 9: 259–277.

Gil Y, Deelman E, Ellisman M, Fahringer T, Fox G, Gannon D et al. (2007) Examining the challenges of scientific workflows. *Computer* 40: 24–32.

Gorissen D, Wendykier P, Kurzyniec D and Sunderam V (2005) Integrating grid information services using JNDI. *6th IEEE/ACM International Workshop on Grid Computing (Grid 2005)*. Seattle, Washington, USA.

Govindaraju M et al. (2003) Merging the CCA component model with the OGSI framework. In *CCGRID '03: Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*. Washington, DC: IEEE Computer Society, p. 182.

Gubała T, Herezlak D, Bubak M and Malawski M (2006) 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)*, Amsterdam, The Netherlands. IEEE Computer Society, p. 12.

Henderson R (1995) Job scheduling under the portable batch system. In Feitelson D and Rudolph L (eds), *Job Scheduling Strategies for Parallel Processing* (*Lecture Notes in Computer Science*, vol. 949). Berlin:Springer, pp. 279–294.

Henning M (2008) The rise and fall of CORBA. *Communications of the ACM* 51: 52–57.

Herráez A (2006) Biomolecules in the computer: Jmol to the rescue. *Biochemistry and Molecular Biology Education* 34: 255–261.

HLA (2010) IEEE standard for modeling and simulation (M and S) high level architecture (HLA) – framework and rules. Technical report IEEE Standard 1516-2010 (Revision of IEEE Std 1516-2000).

Jurczyk P, Golenia M, Malawski M, Kurzyniec D, Bubak M and Sunderam VS (2006) Enabling remote method invocations in peer-to-peer environments: RMIX over JXTA. In Wyrzykowski R, Dongarra J, Meyer N and Wasniewski J (eds), *Parallel Processing and Applied Mathematics, 6th International Conference, PPAM 2005*, Poznan, Poland, September 11–14, 2005, Revised Selected Papers (*Lecture Notes in Computer Science*, vol. 3911). Berlin: Springer, pp. 667–674.

Karonis NT, Toonen BR and Foster IT (2003) MPICH-G2: A grid-enabled implementation of the message passing interface. *Journal of Parallel and Distributed Computing* 63: 551–563.

Kohn SR et al. (2001) Divorcing language dependencies from a scientific software library. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computation*. Portsmouth, VA: SIAM.

Kranzlmüller D, Lucas JM and Öster P (2010) The European Grid Initiative (EGI). In Davoli F, Meyer N, Pugliese R and Zappatore S (eds), *Remote Instrumentation and Virtual Laboratories*. Boston, MA: Springer US, Chapter 6, 61–66.

Krishnan S and Gannon D (2004) XCAT3: A framework for CCA components as OGSA services. In Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS), Santa Fe, NM, pp. 90–97.

Kumfert G, Leek J and Epperly T (2007) Babel remote method invocation. In: *21st International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings*, 26–30 March 2007, Long Beach, CA. IEEE, pp. 1–10.

Kurzyniec D, Wrzosek T, Drzewiecki D and Sunderam VS (2003a) Towards self-organizing distributed computing frameworks: The H2O approach. *Parallel Processing Letters* 13: 273–290.

Kurzyniec D, Wrzosek T, Sunderam V and Slomiński A (2003b) RMIX: A multiprotocol RMI framework for Java. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, Nice, France. IEEE Computer Society, pp. 140—146.

Malawski M, Bartyński T and Bubak M (2010) Invocation of operations from script-based grid applications. *Future Generation Computer Systems* 26: 138–146.

Malawski M, Bartynski T, Ciepiela E, Kocot J, Pelczar P and Bubak M (2006a) An ADL-based support for CCA

components on the Grid. In *CoreGRID Workshop on Grid Systems, Tools and Environments*, Sophia-Antipolis, France.

Malawski M, Bartynski T, Ciepiela E, Kocot J, Pelczar P and Bubak M (2007a) A new approach to supporting component applications on grid. In Bubak M, Turala M and Wiatr K (eds), *Proceedings of Cracow Grid Workshop – CGW'06*, 15–18 October 2006. Krakow, Poland: ACC-Cyfronet AGH, 328–336.

Malawski M, Bubak M, Baude F, Caromel D, Henrio L and Morel M (2007b) Interoperability of grid component models: GCM and CCA case study. In *CoreGRID Symposium* (*CoreGRID Series*). Berlin: Springer, pp. 95–106.

Malawski M, Bubak M, Placek M, Kurzyniec D and Sunderam V (2006b) Experiments with distributed component computing across grid boundaries. In: *Proceedings of the HPC-GECO/CompFrame workshop in conjunction with HPDC 2006*, Paris, France.

Malawski M, Gubala T, Kasztelnik M, Bartynski T, Bubak M, Baude F et al. (2008a) High-level scripting approach for building component-based applications on the grid. In Danelutto M, Fragopoulou P and Getov V (eds) *Making Grids Work: CoreGRID Workshop*, Heraklion, Crete. Berlin: Springer, pp. 307–320.

Malawski M, Harezlak D and Bubak M (2006c) Towards multiprotocol and multilanguage interoperability: Experiments with Babel and RMIX. In Bubak M, Turala M and Wiatr K (eds), *Proceedings of Cracow Grid Workshop – CGW'05*, 20–23 November 2005. Krakow, Poland: ACC-Cyfronet AGH, 266–278.

Malawski M, Kocot J, Ryszka I, Bubak M, Wieczorek M and Fahringer T (2008b) Optimization of application execution in the GridSpace environment. In Gorlatch S, Fragopoulou P and Priol T (eds), *CoreGRID Integration Workshop 2008 – Integrated Research in Grid Computing*, pp. 395–405.

Malawski M, Kurzyniec D and Sunderam V (2005) 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.

Malawski M, Meizner J, Bubak M and Gepner P (2011) Component approach to computational applications on clouds. *Procedia Computer Science* 4: 432–441.

Mayer A, McGough S, Furmento N, Lee D, Newhouse S and Darlington J (2003) ICENI dataflow and workflow: Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, Nottingham, UK, pp. 627–634.

Meizner J, Malawski M, Ciepiela E, Kasztelnik M, Harezlak D, Nowakowski P et al. (2009) Virolab security and virtual organization infrastructure. In Dou Y, Gruber R and Joller JM (eds), *Proceedings Advanced Parallel Processing Technologies, 8th International Symposium, APPT 2009*, Rapperswil, Switzerland, 24–25 August 2009 (*Lecture Notes in Computer Science*, vol. 5737). Berlin: Springer, pp. 230–245.

Morgan RL, Cantor S, Carmody S, Hoehn W and Klingenstein K (2004) Federated security: The Shibboleth approach. *EDUCAUSE Quarterly* 27: 12–17.

Mougin P and Barriolade C (2001) *Web Services, Business Objects and Component Models*. Technical Report, Orchestra Networks.

Murty J (2008) Programming Amazon Web Services – S3, EC2, SQS, FPS, and SimpleDB. Sebastopol, CA: O'Reilly.

NGG Group (2004) Next Generation Grids 2 requirements and options for European Grids research 2005–2010 and beyond. Technical report.

Object Management Group, Inc (2004) *Common Request Broker Architecture Specification, Version 3.0.3*. Available at: http://www.omg.org/cgi-bin/doc?formal/04-03-01

Object Management Group, Inc (2006a) *CORBA Component Model, υ4.0*. http://www.omg.org/technology/documents/formal/components.htm

Object Management Group, Inc (2006b) Deployment and configuration of component-based distributed applications specification, version 4.0. http://www.omg.org/docs/formal/06-04-02.pdf

Ostermann S, Iosup A, Yigitbasi N, Prodan R, Fahringer T and Epema D (2010) A performance analysis of EC2 cloud computing services for scientific computing. In Akan O, Bellavista P, Cao J, Dressler F, Ferrari D, Gerla M et al. (eds), *Cloud Computing* (*Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol. 34(9)). Berlin: Springer, pp. 115–131.

Ousterhout JK (1998) Scripting: Higher-level programming for the 21st century. *Computer* 31: 23–30.

Parlavantzas N, Morel M, Getov V, Baude F and Caromel D (2007) Performance and scalability of a Component-Based grid application. In *Proceedings of IPDPS 2007*, pp. 1–8.

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

Rellermeyer JS, Alonso G and Roscoe T (2007) R-OSGi: distributed applications through software modularization. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, New York, NY. Berlin: Springer, pp. 1–20.

Schatz MC (2009) CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics* 25: 1363–1369.

Schwiegelshohn U, Badia RM, Bubak M, Danelutto M, Dustdar S, Gagliardi F et al. (2010) Perspectives on grid computing. *Future Generation Computer Systems* 26: 1104–1115.

Sloot PM, Tirado-Ramos A, Altintas I, Bubak M and Boucher C (2006) From molecule to man: Decision support in individualized e-Health. *Computer* 39: 40–46.

Sotomayor B, Keahey K and Foster IT (2008) Combining batch execution and leasing using virtual machines. In Parashar M, Schwan K, Weissman JB and Laforenza D (eds) *HPDC '08: Proceedings of the 17th International Symposium on High Performance Distributed Computing*. New York: ACM Press, pp. 87–96.

Streit A, Erwin D, Lippert T, Mallmann D, Menday R, Rambadt M et al. (2005) Unicore – From project results to production grids. *Advances in Parallel Computing* 14: 357–376.

Thain D, Tannenbaum T and Livny M (2005) Distributed computing in practice: the Condor experience. *Concurrency and Computation – Practice and Experience* 17: 323–356.

van Nieuwpoort R, Maassen J, Wrzesinska G, Hofman RFH, Jacobs CJH, Kielmann T et al. (2005) Ibis: a flexible and efficient Java-based grid programming environment. *Concurrency – Practice and Experience* 17: 1079–1107.

Van Nieuwpoort RV, Wrzesińska G, Jacobs CJH and Bal HE (2010) Satin: A high-level and efficient grid programming model. *ACM Transactions on Programming Languages and Systems* 32: 3.

Vecchiola C, Pandey S and Buyya R (2009) High-performance cloud computing: A view of scientific applications. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, pp. 4–16.

Vogels W (2003) Web services are not distributed objects. *IEEE Internet Computing* 7: 59–66.

Wilde M, Foster I, Iskra K, Beckman P, Zhang Z, Espinosa A et al. (2009) Parallel scripting for applications at the petascale and beyond. *Computer* 42: 50–60.

Wilson N and Johnston R (2000) Modelling gold clusters with an empirical many-body potential. *European Physical Journal D* 12: 161–169.

Witten IH and Frank E (2005) Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems). Waltham, MA: Morgan Kaufmann.

Zhao Z, Belloum A and Bubak M (2009) Special section on workflow systems and applications in e-Science. *Future Generation Computer Systems* 25: 525–527.

Author's Biographies

Maciej Malawski, PhD in computer science and MSc in computer science and in physics. Researcher and lecturer at the Institute of Computer Science, AGH and at ACC Cyfronet, AGH. Postdoc at Center for Research Computing, University of Notre Dame, USA. Coauthor of over 50 international publications including journal and conference papers, and book chapters. Involved in the EU IST ViroLab project, where he was the leader responsible for the middleware task and for contacts with external users. Responsible for the Virtual Laboratory developed in the PL-Grid project. His scientific interests include parallel computing, grid systems, distributed service- and component-based systems, and scientific applications.

Marian Bubak, PhD, is an adjunct at the Institute of Computer Science, AGH, a staff member at the ACC Cyfronet, AGH, and the Professor of Distributed System Engineering at the Informatics Institute of the Universiteit van Amsterdam. His research interests include distributed and grid systems for scientific simulations. He has co-authored about 230 papers. He led the architecture team of the EU IST CrossGrid Project, he was the Scientific Coordinator of the K-WfGrid Project and a member of the Integration Monitoring Committee of CoreGRID. He has served as a program committee member, chairman and organizer of several international conferences (HPCN, Physics Computing, EuroPVM/MPI, SupEur, HiPer, ICCS, HPCC, e-Science 2006); he is co-editor of 17 proceedings of international conferences.

Tomasz Gubała, MSc in Computer Science, worked for the Computational Science Department at the University of Amsterdam, the Netherlands, as a scientific programmer and computer science research assistant. He was involved in the major EU-funded project ViroLab as a chief designer of the virtual laboratory for infectious diseases. He is an external PhD student at the Department of Computational Science at the University of Amsterdam, he works at the ACC Cyfronet in Krakow as a scientific programmer, and also applies his research as a part-time commercial solutions developer. His main scientific interests are semantic modeling of application domains, semantic integration of tools, distributed computing, and services for e-Science.