**Journal of Grid Computing manuscript No.**
(will be inserted by the editor)

# ServiceSs: an interoperable programming framework for the Cloud

**Francesc Lordan · Enric Tejedor · Jorge Ejarque · Roger Rafanell · Javier Álvarez · Fabrizio Marozzo · Daniele Lezzi · Raül Sirvent · Domenico Talia · Rosa M. Badia**

Received: date / Accepted: date

**Abstract** The rise of virtualized and distributed infrastructures has led to new challenges to accomplish the effective use of compute resources through the design and orchestration of distributed applications. As legacy, monolithic applications are replaced with service-oriented applications, questions arise about the steps to be taken in order to maximize the usefulness of the infrastructures and to provide users with tools for the development and execution of distributed applications. One of the issues to be solved is the existence of multiple cloud solutions that are not interoperable, which forces the user to be locked to a specific provider or to continuously adapt applications.

With the objective of simplifying the programmers challenges, ServiceSs provides a straightforward programming model and an execution framework that helps on abstracting applications from the actual execution environment. This paper presents how ServiceSs transparently interoperates with multiple providers implementing the appropriate interfaces to execute scientific applications on federated clouds.

**Keywords** Cloud Computing · Programming Models · Interoperability · Standards

F. Lordan, E. Tejedor, J. Ejarque, R. Rafanell, J. Álvarez, D. Lezzi, R. Sirvent, R. M. Badia
Department of Computer Sciences,
Barcelona Supercomputing Center (BSC-CNS)
Barcelona - Spain
E-mail: francesc.lordan@bsc.es, enric.tejedor@bsc.es, jorge.ejarque@bsc.es, roger.rafanell@bsc.es, javier.alvarez@bsc.es, daniele.lezzi@bsc.es, raul.sirvent@bsc.es

R. M. Badia
Artificial Intelligence Research Institute (IIIA),
Spanish Council for Scientific Research (CSIC)
Barcelona - Spain
E-mail: rosa.m.badia@bsc.es

F. Marozzo, D. Talia
DIMES, University of Calabria, Rende (CS), Italy
E-mail: fmarozzo@deis.unical.it, talia@deis.unical.it

D. Talia
ICAR-CNR, Rende (CS), Italy

## 1 Introduction

Cloud computing has emerged in the recent years as an answer to the needs of the society for computing services on demand. With this paradigm, IT users have access to computing services reducing the cost of ownership and operation to the minimum. What is more, they can establish SLAs with the providers that guarantee their QoS requirements. Well known examples of commercial offerings of Cloud Computing are the Amazon Elastic Compute Cloud (Amazon EC2) [1], Microsoft Windows Azure [14], or Google App Engine [10]. On the open source side, several solutions are available to to build private clouds, like OpenNebula [53], Eucalyptus [48], OpenStack [51] or EMOTIVE Cloud [31].

Initially adopted in the business sectors due to economical choices in the design of IT departments, cloud has started to have wide acceptance also in science sectors. The scientific community adopted grid computing as a paradigm suitable to run their applications, with features such as sharing and volunteering at a best effort policy being appealing to them. The appearance of cloud computing has somehow eclipsed the grid and the scientific community is looking at the cloud as an alternative computing paradigm, although supercomputing is still considered as well for those applications that have specific high performance computing requirements.

Writing an application that uses resources of a distributed environment is not as easy as writing a sequential application, and requires the programmer to deal with a number of technological concerns. One of the issues to be solved in this process is the existence of multiple cloud solutions that are not interoperable due to the current difference between the individual vendor approaches. The lack of interoperability has to be solved in order to allow applications and services to run on federated infrastructures without having to continuously adapt the code. Another important property of cloud computing that poses a requirement in the design of the runtime is the elastic scaling, i.e. the capability to automatically provision (and de-provision) resources on demand as needed by users and applications load.

The design of a programming framework that allows the execution of scientific applications on top of virtualized infrastructures is a pressing requirement for those scientists interested in exploiting the capabilities of clouds.

Service Superscalar (ServiceSs) is a programming framework that aims at solving the previous issues easing the development of applications (i.e. parallel applications, business or scientific workflows, compositions of services mixing services and code, etc.) and their execution on distributed environments. From now on, in this paper, we will refer indistinctly to complex services or applications when describing implementations done with ServiceSs. The reader must consider that ServiceSs can be used for all these different types of applications.

This paper focuses on two aspects of the proposed framework, the programming model and the interoperability features of the runtime. The framework implements a task-based programming model that allows applications to be written following a sequential paradigm and without need of a specific API, leaving to the runtime the responsibility to execute the code detecting data dependencies between tasks and exploiting the inherent parallelism of the sequential code.

The other important feature of the ServiceSs programming framework is the capability to execute the applications transparently with regards to the underlying infrastructure. The availability of a different connector, each implementing the specific cloud provider API, makes possible the run of computational loads on multi cloud environments without the need of code adaptation, providing scaling and elasticity features and allowing to adapt the number of available resources to the actual execution needs.

This paper presents the details of the ServiceSs programming model and the programming of composite workflows in sections 2 and 3. Section 4 describes a component that allows the deployment and execution of ServiceSs applications. Section 5 details how the composed services are orchestrated by the runtime and section 6 analyzes how ServiceSs achieves interoperability through specific connectors. Section 7 presents results of the evaluation of the proposed framework through the execution of ServiceSs applications on different cloud testbeds and analyzing the overhead introduced by the runtime. Section 8 describes the related work and section 9 concludes the paper.

## 2 ServiceSs introduction: an interoperable programming framework

### 2.1 Solutions to current challenges

In order to overcome the challenges presented previously, we propose a programming model that simplifies the development and execution of applications in cloud infrastructures. Our proposal has as a starting point the COMPSs programming model [54]. We have adapted the theoretical foundations of COMPSs to the particularities of cloud infrastructures, creating ServiceSs [55]. We summarize these foundations in the next paragraphs, although we encourage the reader to visit the mentioned references for more specific details.

The most important idea behind COMPSs is the possibility of developing a sequential application that can be run in parallel in the underlying infrastructure. This idea is based on how superscalar microprocessors execute instructions in an out-of-order manner [35]. Instructions to be executed in the processor have input and output registers, and in order to guarantee that an out-of-order execution will produce a correct result, data dependencies between instructions need to be detected. In this analogy with superscalar microprocessors applied to a different scale, the instructions correspond to method calls inside an application, and the registers correspond to the data that a method is reading or writing. COMPSs runtime automatically creates a workflow that describes data dependencies between method calls. Besides, as it is done in superscalar processors, data renaming techniques can be applied in order to eliminate false data dependencies (WaR, WaW). In summary, with this idea we enable sequential applications to run in parallel in a distributed infrastructure.

COMPSs applications are implemented in a sequential way, and without APIs that deal with the infrastructure or with duties of parallelization and distribution (synchronizations, data transfer, ...). This is very important to ensure to provide a unique and simple programming interface to create applications. This means, on the one hand, that the application will not be based on a specific API to express the interaction with the

infrastructure, thus avoiding vendor lock-in and lack of portability of applications. On the other hand, we are adopting sequential programming as the main programming paradigm, which is the most easy paradigm to be offered to end users, therefore achieving an easy way for users to program applications. Users do not need to think of how their program is going to be run in the infrastructure, because the COMPSs runtime will take care of the actual execution of the application on a specific infrastructure. Instead, users only need to focus on their specific domain of knowledge to create a new program that will be able to run in the cloud.

Another key aspect of providing a cloud-unaware programming model is that programs can be developed once and run in multiple clouds without changing the implementation. This is very important when portability between clouds must be achieved. In ServiceSs, the programmer is freed from having to deal with the specific cloud details, because ServiceSs runtime will be in charge of it. As discussed later in this article, the runtime follows a plug-in approach to deal with several cloud frameworks, hiding this burden to the end user and enabling interoperability.

In addition, one of the main players in the cloud architecture is the Web Service. The cloud is widely used to deploy services, since it offers a very dynamic environment suitable for them. Thus, our interest is to be able to support Web Services when creating applications in the cloud, and not only in an isolated manner, but Web Services that interact between them (typically described by a workflow). This is known in the literature as orchestration of services. In ServiceSs, we have included the capability of orchestrating workflows composed of services mixed with regular methods (pieces of code not intended to be services). Both services and methods can be ServiceSs tasks and be part of a ServiceSs application workflow. An application published as a service and contains calls to service and method tasks will be referred to as a *composite service*.

Taking advantage of the knowledge the ServiceSs runtime has at application level (the workflow), ServiceSs is able to determine at any moment if reserving more resources in the cloud could pay off or not. This is mainly because of the data dependencies related to the application that is being executed. If there are many tasks ready to be executed (no data dependencies to be solved), new resources could be useful. In this article we will see how adequate is the cloud to play with the elasticity of resources dedicated to an application, when workflows are used as the main knowledge to make decisions.

The class of applications targeted by ServiceSs are based on algorithms that may be easily split in calls to computational methods or services, usually coarse-grained, loosely coupled, with low requirements on communications and whose parameters are files, objects, arrays or primitive types. ServiceSs is typically adopted to port legacy applications used mainly in the scientific domain and whose execution patterns follow high-throughput parallelism, data flows and many-task coordination models. Workflows are implemented in scientific applications, as for example in the bioinformatics area, dealing with data and going through different stages with different levels of concurrency. These problems can be split into several individual sub-problems thus needing aggregation steps. In this case, applications need data synchronization and automatic staging-in and out of files, according to their dependencies. These workflows may require also to use control-flow features for the coordination of these sub-problems.

### 2.2 Summary of the solution

ServiceSs does not only provide a programming model, but the framework is complemented with a set of platform tools which ease: the ServiceSs applications implementation by means of an Integrated Development Environment (IDE); the application deployment in distributed infrastructures by means of the Programming Model Enactment Service (PMES); and the monitoring of executions by means of the Monitoring and Tracing tools. Figure 1 shows a diagram with all the tools composing the framework, and their correspondence to a service lifecycle.

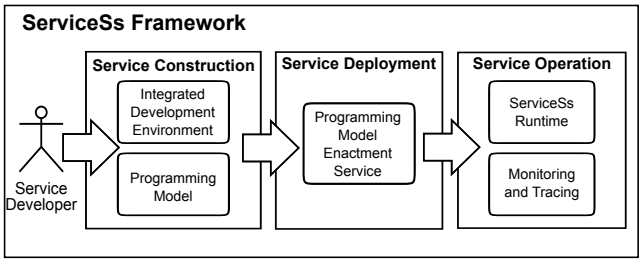



**Fig. 1** ServiceSs Framework Overview.

The programming model syntax enables the easy development of cloud applications as composite services. A composite is written as a sequential program from which other services and regular methods are called. Therefore, composites can be hybrid codes that reuse functionalities wrapped in services or methods, adding some value to create a new product that can also be published as a service. Besides, all the information needed for data-dependency detection and task-based parallelization is contained in a separate annotated interface.

Above all, the cloud is kept transparent to the programmer as no information about aspects like deployment or scheduling has to be described in the code.

The model can be defined as task-based and dependency-aware. In a first step, users select the set of services and methods called from the composite that they intend to run as tasks in parallel on the available resources. The task selection is done by means of a interface which declares those services/methods, along with some metadata in the form of annotations [11]. One of these annotations is used to state the direction of each task parameter (input, output or in-out); with this information the ServiceSs runtime discovers, at execution time, the data dependencies between tasks and dynamically builds a dependency graph.

From this point on, the composite service will be referred to as Orchestration Element (OE). Those services and methods, invoked from the OE and selected as candidates to be executed in the cloud, will be called Core Elements (CE), and the interface where they are chosen will be the Core Element Interface (CEI). Thus, to clarify the terminology for the rest of the article: a CE can be a coded method (as in COMPSs) or a Web Service. When a CE is invoked we talk about a *task*, which is also a node in the application workflow. So, a *task* is effectively an instance to an invocation of a CE. And finally, when the task is executed in a particular resource, we will use the term *job*, that is the final incarnation of the CE (i.e. a piece of code executed in a resource).

On the other hand, the offered platform services are composed by the IDE, the PMES and the Monitoring and Tracing tools. The IDE programming environment for developers, acts as a showcase of the programming model. Its implementation is based on the Eclipse platform [5], a very popular development environment on programmers world. Targeting Eclipse allows us to achieve visibility to a wide audience of developers as potential users of the programming model.

The PMES provides a standard interface enabling the deployment and execution of the new programmed service on different types of infrastructures. The IDE directly invokes PMES to deploy the virtual appliances related to a new programmed service in the different infrastructures that will be executing it, thus acting as a single contact point to deal with the heterogeneity of the underlying platform middlewares.

The Monitoring and Tracing tools provide online information about the execution of the service, allowing end users to visualize the existing data dependencies between service components, and what components have been executed or are pending in a particular moment. Besides, this component is able to generate low-level trace files of the execution (i.e. detailing CPU time, idle time, etc.) that can be analyzed in order to precisely determine the performance and identify possible bottlenecks.

## 3 Service Construction

As introduced in section 2, the Service Construction layer of the ServiceSs framework is composed of the Programming Model (PM) and the IDE. On one side, the PM provides a syntax to implement applications for cloud infrastructures without using any kind of provider API and offers the capability to port the application and distribute the service components across multiple cloud providers. On the other side, the IDE provides a user-friendly environment to guide the users during the implementation and building phases. The next subsections provide more details about the PM syntax and the IDE component.

### 3.1 Programming Model Syntax

Section 2 already introduces the main concepts behind the ServiceSs Programming Model. The application developer writes a sequential code for the application and, then, selects which of its methods becomes a CE by declaring them in the CEI. Any ServiceSs application can be composed of two different kinds of CE: Method CE and Service CE. On the one hand, Method CEs are regular methods of the application selected to be run remotely. To pick a method CE, the programmer declares the method in the CEI adding the *@Method* annotation indicating the implementing class.

On the other hand, Service CEs correspond to SOAP Web Service operations described in WSDL documents. To select a SOAP operation as a CE, the developer declares the service operation accompanied by the *@Service* annotation describing the service details (namespace, service name and service port). The location of the service is not included in the CEI, but in the runtime configuration that actually decides which server runs the task; thus, the programming model syntax remains completely unaware of the underlying infrastructure.

For the runtime system to determine the dependencies between the different CE composing the application, the programmer has to state how the CE operates on each data it accesses, i.e. its parameters. For this purpose, the programmer adds some annotations(*@Parameter*) to the CE declaration in the CEI specifying for each piece of data its type and its direc-

tion (in, out, inout) depending on the operation performed on it (reading, creation, modification).

Finally, by adding the *@Constraints* annotation, the programmer can impose a set of hardware and software constraints to be fulfilled by any resource hosting a task execution of a given CE.

The implementation of the ServiceSs runtime has been performed using the Java language and, because of this fact, it becomes a natural programming language to create ServiceSs applications (like the example contained in Figure 2). Nevertheless, other programming languages such as C, Python or Scala are supported to create new ServiceSs applications or port already existing ones.

Subfigure 2(a) contains an example of annotated interface with two CEs declared: *update* and *sampleService*. *Update* corresponds to a method CE implemented in the *sample.Example* class as line 1 details. The declaration also describes the accesses to its three parameters: *option*, an integer that is read (lines 6-7); *value*, an object that is modified (lines 8-9); and *log*, a file created along the task execution (lines 10-11). The *@Constraints* annotation on line 3 restricts its execution to resources with more than 4 cores and at least 1GB of physical memory.

The other CE defined in the *SampleCEI* corresponds to a SOAP web service operation in the service *sampleService* detailed in the *@Service* annotation in lines 14 and 15. The *sampleOperation* CE operates on two pieces of data: a *Query* object read by the service and the return value of the operation: a *Reply* object. The classes involved in service operations, such as *Reply* and *Query*, can be generated directly from the WSDL of the service.

Once the CEI is completed, the CEs defined in it can be combined in Orchestration Elements. The body of an OE is programmed as a sequential code, without using any API or specific syntax constructions. Subfigure 2(b) depicts a sample OE composed by CEs declared on the subfigure 2(a) CEI. A regular method is marked as OE by adding an *@Orchestration* annotation to it as shown in line 1. The CEs are invoked as regular methods; the ServiceSs runtime is in charge of replacing the method invocation with the creation of a new task, managing its dependencies and execution on top of the available infrastructure.

The following paragraphs point out some key operations used in the sample OE code and summarizes the runtime behavior in each case. [55] includes further details about the task detection, the management of data dependencies and the scheduling of tasks.

```
1  public interface SampleCEI {
2      @Method(declaringClass = "sample.Example")
3      @Constraints( processorCPUCount = 4,
4                    memoryPhysicalSize = 1.0f)
5      void update(
6          @Parameter(direction = IN)
7          int option
8          @Parameter(direction = INOUT)
9          Reply value
10         @Parameter(type=FILE, direction = OUT)
11         String log
12     );
13
14     @Service(namespace = "http://servicess.com/example",
15        name = "SampleWS", port = "samplePort")
16     Reply sampleService(
17         @Parameter(direction = IN)
18         Query query
19     );
20
21 }
```

(a) Sample Core Element Interface. The *update* method is designated as a method CE implemented in the *sample.Example* class. It accesses three pieces of data: reads an integer, modifies an $R$ object and creates a file; its execution is restricted to resources with more than 4 cores and 1GB of memory. The other CE, (*sampleOperation*) is declared as Service CEs linked to the *samplePort* port of the *sampleWS* service that reads a *Query* object and produces a *Reply* object.

```
1  @Orchestration
2  public void SampleOE(String[] args) {
3
4      Example e = new Example();
5
6      Query query = new Query(args[0]);
7      Reply reply = sampleOperation(query);
8
9      e.update(3,reply,"log.txt");
10
11     System.out.println(e.getValue());
12 }
```

(b) Sample Orchestration Element using the CE presented in Figure 2(a).

**Fig. 2** Sample application code written in Java.

*Service CE invocation* Lines 6-7 show a simple example of how to call a service. The *sampleOperation* represents a SOAP operation that receives a *Query* instance and produces a *Reply*. The call to *sampleOperation* by the OE is performed on a local method representative of the service operation (generated beforehand for compiling purposes). This representative is never executed because the runtime replaces the actual call by a task creation. Since the *query* object is generated by the OE, the task has no dependencies and its execution is scheduled immediately.

*Method CE invocation* As in any normal Java application, there are two ways to invoke a method: statically or in a class instance. Line 9 contains an example of a non-static method invocation where the OE invokes the *update* method on the *Example* instance (e). Since $e$ has been declared and initialized in line 4 and no other task modifies it, the runtime does not detect any depen-

dency due to this access. However, the task accesses the *Reply* instance generated by the task created by line 7. The runtime waits until the *update* task is free of dependencies, i.e. the *sampleOperation* task ends producing the reply instance, to schedule the task. In addition to its parameters, any operation can also access two more pieces of data: its callee ($e$) and the return value. Since the callee and return value are objects and their directions are implicit (inout and out respectively), the programmer does not need to describe them in the CEI.

*Synchronization in the OE* Any data produced or modified by a task can be accessed later by the OE as depicted in line 11. From the point of view of the programmer, the access to this data is not different whether it has been written by a previous task or not. In order to keep the coherence of the values and guarantee the correct execution of the application, when a task modifies a preexisting object or creates a new one which is accessed later by the OE, a synchronization is needed to fetch the proper value. From that moment on, the main code execution is paused, i.e. no more tasks are generated, until the value is accessible.

### 3.2 Integrated Development Enviroment

The ServiceSs Integrated Development Environment provides a graphical interface for facilitating the construction of services and applications following the programming model syntax, described in section 3.1. The IDE is implemented as an Eclipse [5] plug-in which extends the Java development tools offered by the Eclipse core with a Service Editor (Figure 3) and a set of wizards and actions to implement and build ServiceSs applications in an easy way. The Service Editor guides the developers during the process of defining, implementing and building the Orchestration and Core Elements according to the Programming Model syntax. It is composed of two tabs: the *Implementation* tab which provides an overview of the service implementation status showing the Orchestration and Core Elements defined for each class, and the *Build and Deploy* tab which provides a graphical interface to build and deploy the service once its implementation has been finished. From the *Implementation* tab, developers can execute different wizards to perform the creation of Orchestration and Core Elements from scratch as well as from existing software such as jar libraries, binaries and web services. This wizards also automatically creates the Core Element Interface in a transparent way for the developer. On the other hand, developers can use the *Build* section to perform the building action which compiles, instruments and packages the different service elements in order to have the ServiceSs application ready for deployment and execution. Finally, the IDE provides a deployment widget which allows developers to deploy the ServiceSs application either in the local host, to test and debug the application, or in a production cloud infrastructure through the Programming Model Enactment Service.

## 4 Service Deployment

The transparent deployment of ServiceSs applications on cloud infrastructures is delegated to the PMES [38] PaaS component. The PMES exposes the needed operations to ServiceSs IDE dealing with the intricacies of deployment and contextualization operations, of the installation of the application packages and of the required libraries and of the monitoring processes.

### 4.1 PMES Interface

The PMES interface implements the OGF HPC-BP [20] profile that includes the adoption of the Basic Execution Service (BES) [8] specification to define the set of operations available to instantiate an application and of the Job Submission Description Language (OGF JSDL) [24] used to provide the details of the deployment and execution operations.

The BES interface provides operations to update, monitor and terminate deployments and to request applications status at runtime, which are: *Pending, Running* and *Finished* states for proper deployment situations, and *Failed* or *Cancelled* for terminated ones, as defined by the standard BES states-model. Moreover, more detailed information about ServiceSs realtime monitoring and average cloud workload can be also obtained. Thanks to the adoption of standards other existing BES-compliant clients, as for example Unicore [37], can be used to instantiate computational loads on this management system, being PMES a key component on ServiceSs ecosystem.

### 4.2 PMES Implementation

The PMES implements, on top of the Simple API for Grid Applications (SAGA) [32], a manager which deals with the deployments lifecycle. The use of SAGA enables the deployment on a set of different infrastructures and middlewares such as PBS, TORQUE, Amazon EC2, etc.

Figure 4 depicts the ServiceSs-PMES architecture and its main components:

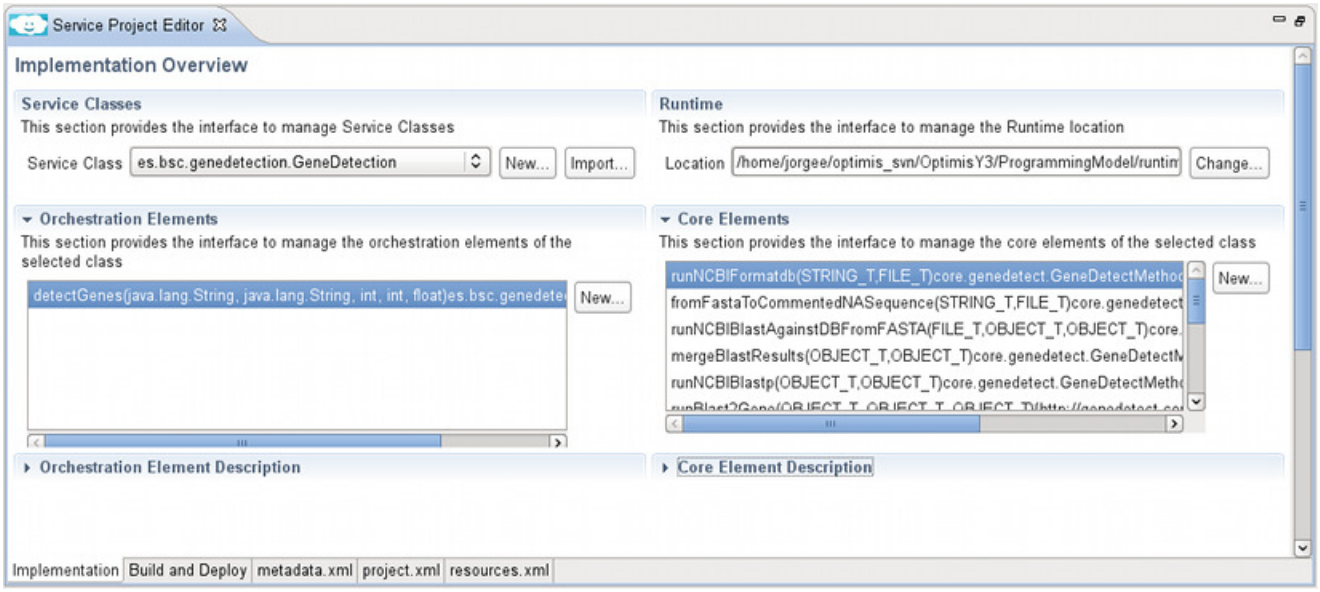



**Fig. 3** IDE Service Editor screenshot

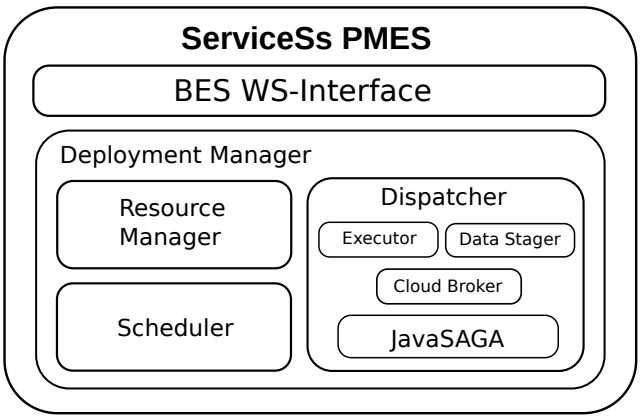



**Fig. 4** The ServiceSs-PMES architecture.

– *Resource Manager:* controls a pool of usable resources notifying the scheduler of their availability for a particular deployment request.
– *Scheduler:* checks with the Resource Manager the availability of the required resources to deploy the application or service and forwards the request to the Dispatcher component. If no resources are available the operation is enqueued and handled according to the chosen scheduling policy.
– *Dispatcher:* handles the deployment cycle interacting with the following components:

1. *Cloud Broker:* requests virtual machines to a cloud provider and deploys the ServiceSs application package as well as the needed applications/libraries.
2. *Data Stager:* deals with service/application data transfers (such as logs, periodically-updatable files, ...) moving the required input/output data from the user-defined storage location to the local disk of the VM. Several back-ends are supported such as Cloud Data Management Interface (CDMI) [40], FTP, etc.
3. *Executor:* initialize the application once it has been deployed in the VMs.
4. *JavaSAGA:* the Java implementation of the SAGA specification is used as an interface to different communication protocols. In PMES, persistent SSH connections are established with every deployed virtual machine.

## 5 Service Operation

Once a service is deployed on top of a cloud infrastructure, it becomes ready to start receiving execution requests. When such a request arrives, the Service Operation layer of the framework runs the service and monitors its execution. On the one hand, this third layer of the framework provides the user with a runtime system that orchestrates the execution of the tasks composing the application on top of a resource pool. On the other hand, it allows the user to inspect the progress of the service execution and profile it in order to make a proper performance analysis. The following subsections give more details about the components of the operation layer.

### 5.1 Runtime System

The ServiceSs syntax, described in section 3.1, allows developers to compose parallel applications and services being totally unaware of the infrastructure and the parallelism details. The sequential code is executed, and the ServiceSs runtime intercepts the CE invocations replacing them by calls to the runtime that creates new asynchronous tasks. Also accesses to task data from the OE are instrumented, so that the runtime fetches the proper values of the data from the remote resource where they were generated (synchronization).

In order to support several programming languages, as introduced in section 3.1, the tools and the processes used for implementing these interceptions code are bound to each specific language. For instance, Java codes, like the one presented on Figure 2, are modified before the OE is executed. We developed a custom Java class loader that uses Javassist [12], a library for Java class editing, to create a new class with the necessary code to create tasks and trigger synchronizations. This modified class replaces the original one and runs as a normal Java class with methods that include calls to the runtime interface.

The runtime architecture is composed by two main components: the Task Processor (TP) and the Task Dispatcher (TD). The Task Processor is the front-end of the runtime. It receives calls from the application code and analyses each data access looking for the data dependencies with previous tasks. When a task is free of dependencies, the TP sends it to the back-end of the runtime, the Task Dispatcher, which schedules the execution of the tasks and submits a job to a free resource. Figure 5 depicts an overview of the architecture of the runtime. The following paragraphs go into detail of the functionalities of each component that appears on it.

The Task Processor leverages on the Task Analyzer (TA) and the Data Info Provider (DIP) to check the task dependencies. The DIP is responsible for applying the renaming technique by creating a new version of each data every time a task writes on it. By keeping track of all data versions, the DIP assigns a unique renaming to each version. On the other hand, the Task Analyzer keeps the information related to the task dependencies up to date by storing them in the form of a directed acyclic graph where tasks are represented as nodes and arcs depict the dependencies between tasks. When one node of this graph has no longer any direct predecessor, the task represented by that node is free of dependencies and it is submitted to the Task Dispatcher.

Regarding the Task Dispatcher, scheduling is managed by the Task Scheduler (TS) in cooperation with the Scheduler Optimizer (SO) and the Resource Manager (RM). As soon as the Task Dispatcher receives the task, the Task Scheduler picks a resource to host its execution using a basic algorithm taking into account the features of each resource (stored by the RM), the constraints of the invoked CEs and the number of tasks that can run simultaneously in each resource (slots). Besides, in order to improve the load balancing, the Scheduler Optimizer dynamically checks and modifies, if necessary, the TS decisions. Although both scheduling policies have an important impact on the application performance, we do not describe them in this paper since this kind of algorithms is out of the scope of this paper context.

Once the TS decides that is the proper time to execute a task, it communicates its decision to the Job Manager (JM), which actually manages its execution. On the one hand, in case of dealing with a task corresponding to a method CE, the JM submits it to the selected resource and monitors its execution. Before running the task, the JM must ensure that the remote resource has all the data required to execute it, i.e. all the input values missing on the resource must be transferred there. The Job Manager delegates the data management to the File Transfer Manager (FTM). On the other hand, if the JM deals with a service CE task, the service is invoked from the JM using a dynamic client generated through Apache CXF configured with the data extracted from the *@Service* annotation properties (see section 3.1). Since some data required by the task might be the result of a method CE task, the input values for the invocation can be located in other resources; again, the JM delegates the FTM to obtain these values.

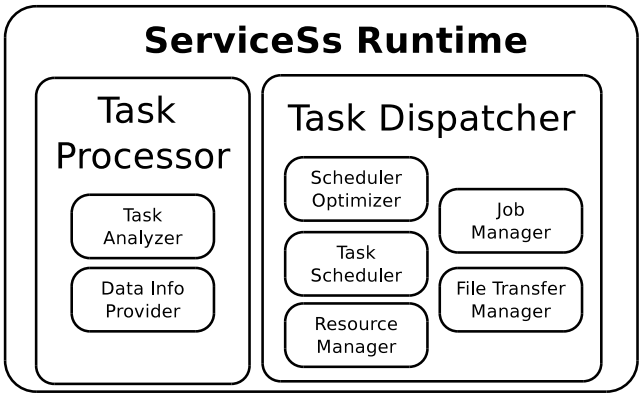



**Fig. 5** Architecture of the ServiceSs runtime.

#### *5.1.1 Job Management*

The interaction between the ServiceSs runtime and the computational resources is a critical point in the design of the framework interoperability due to the wide variety of existing protocols and middlewares. The ServiceSs runtime leverages JavaGAT [13] to enable the interaction with the resources to execute the application tasks whether they are physical resources or VMs deployed in the cloud. JavaGAT is a toolkit which offers a uniform interface to interact with computational resources; the calls to its API are redirected to a specific adaptor which operates through a given middleware with some remote resource. ServiceSs uses JavaGAT API for two capabilities: requesting data transfers and submitting tasks to remote resources.

#### *5.1.2 Resource Management*

Another important feature of the ServiceSs runtime is its ability to exploit the cloud to adapt automatically the size and capabilities of the resource pool to the current workload. Depending on the amount of tasks to execute and their requirements the runtime can turn to one cloud provider or another to change the amount of computational resources deployed on their servers. When the runtime notices a lack of resources, it asks for them to the cloud provider who offers the kind of resources that fits better to the application needs. Symmetrically, when it detects that some resources are no longer profitable, it orders their destruction.

The component in charge of detecting the lack/excess of resources is the Scheduler Optimizer. By considering the amount of tasks waiting for an available resource, their hardware and software constraints (provided by the developer in the application CEI) and some historical data about previous tasks execution, the SO quantifies the amount resources required by the application and asks the Resource Manager to interact with the cloud providers to get/free the proper amount of virtual resources.

In order to simplify the Resource Manager code and make it interoperable with different cloud providers, we have defined a simple and generic interface to manage the creation of additional resources, free already deployed ones and query the providers about cost details. The implementations of this interface are called *connectors* and each of them manages the actual interactions with the cloud providers that offer the same API.

Many cloud providers restrict the features of their resources by offering some predefined resource templates (e.g. small, medium, large) and they contextualize them with some software and files by loading some pre-created image or some configuration file. The application user specifies which providers are available and the usage of each one by means of a configuration XML file. There, he specifies for each provider which connector has to be used for interacting with it, the offered images and templates, the range of resources that can be deployed simultaneously or specific connector details such as the authentication information or the provider endpoint.

### 5.2 Monitoring and Tracing

In addition to the functionality explained above, the ServiceSs runtime components also collect different information about how the application execution is progressing. It stores the Core Element invocations by the current application execution, data dependencies detected between these invocations, which Core Elements have already been executed, which are currently working and which are still pending. Based on this information, the ServiceSs framework provides a Monitoring tool to visualize: the data dependency graph generated at runtime, which part of it is currently running and how the current resources are used to execute this graph.

At the end of each Core Element execution or file transfer, the ServiceSs runtime also creates usage records [42]. This usage records contain information about the resource involved in the Core Element execution, the source and destination resources in data transfers, as well as the start and end time of each operation. Once the application finishes, all these usage records can be processed by the Tracing tool in order to perform a postmortem reconstruction of the application execution across the different cloud resources. This reconstruction can be visualized by tools such as Paraver [30] in order to detect bottlenecks and unbalanced parts of the application which could be fixed to increase the application performance.

## 6 Exploiting Interoperability

In ServiceSs, there exist two mechanisms that enable interoperability between different cloud vendors. On the one hand, the ServiceSs runtime is able to interact with diverse IaaS providers by means of pluggable connectors. Each connector implements the communication of the runtime with a given provider, e.g. to create or destroy a VM. On the other hand, ServiceSs also features adaptors that manage jobs and data transfers in the cloud in two scenarios: first, in virtual resources obtained from an IaaS provider; second, by invoking a particular PaaS API.

The next subsections explain how ServiceSs achieves interoperability both at IaaS and PaaS level, and provide examples of the interaction of ServiceSs with different cloud providers.

## 6.1 IaaS Interoperability

A feature shared among all the IaaS providers is the SSH protocol used to access the VM instances requested by their customers. ServiceSs leverages the JavaGAT SSH as a default adaptor for job management and data transfer. Although the market has reached an agreement on the VM access, the IaaS providers have not agreed on a standard API for the VM instance management. For this reason, the ServiceSs runtime defines a common interface whose implementations (the connectors) translate a set of abstract operations into calls of the specific API offered by the provider. Such operations include the ones to *create* and *release* VMs, used by ServiceSs runtime when decides to vary the computing pool. Besides, connectors also implement the operations to get the *cost* per hour and *creation time* of a certain VM in the provider.

The next subsections describe two examples of IaaS connectors implemented by the ServiceSs runtime.

### *6.1.1 Open Cloud Computing Interface (OCCI) connector*

One solution provided by the ServiceSs framework for interoperating with different clouds is the OCCI [16] connector. It implements the operations defined in the ServiceSs runtime connector using a subset of the OCCI v1.0 interface to create Compute and Network resources. As a Compute resource descriptor for this OCCI version, the connector uses the OVF, which provides a standard way to describe the characteristics of the VMs. This connector provides the ServiceSs runtime with the availability to interoperate with all the cloud providers whose front-ends are compatible with this OCCI + OVF standard format. Two examples of this are EMOTIVE Cloud, which offers it by means of the Dynamic Resource Provisioning (DRP) service, and OpenNebula, providing it through the OVF4ONE [18] service.

### *6.1.2 Amazon EC2*

The ServiceSs runtime also features a connector to interact with the Amazon Elastic Compute Cloud (EC2), the public cloud solution that is currently dominating the IaaS market.

Amazon EC2 offers a well-defined pricing system for VM rental. A total of 8 pricing zones are established, corresponding to 8 different locations of Amazon datacenters around the globe. Besides, inside each zone, several per-hour prices exist for VM instances with different capabilities. The ServiceSs EC2 connector stores the prices of standard on-demand VM types (micro, small, medium, large and extra large) for each zone.

When the ServiceSs runtime chooses to create a VM in Amazon, the EC2 connector receives the information about the requested characteristics of the new VM, namely the number of cores, memory, disk and architecture (32/64 bits). According to that information, the connector tries to find the VM type in Amazon that better matches those characteristics and then requests the creation of a new VM instance of that type.

Once an EC2 VM is created, a whole hour slot is paid in advance; therefore, it makes sense to keep the VM alive at least during such period. For this reason the EC2 connector saves VMs for later use. When the task load decreases and the VM is no longer necessary, the connector puts it aside if the hour slot has not expired yet, instead of terminating it. After that, if the task load increases again and the EC2 connector requests a VM, first the set of saved VMs is examined in order to find a VM that is compatible with the requested characteristics. If one is found, the VM is reused and becomes eligible again for the execution of tasks; hence, the cost and time to create a new VM are not paid. A VM is only destroyed when the end of its hour slot is approaching and it is still in saved state.

## 6.2 PaaS interoperability

As in the IaaS case, ServiceSs also manages PaaS resources by means of connectors that interact with a particular PaaS API. More precisely, the ServiceSs runtime manages PaaS offerings by abstracting the computing platform resources as a single logical machine, where multiple jobs can be executed at the same time. In contrast to the IaaS case, where new resource templates are instanced, for PaaS the runtime enlarges or shrinks the amount of computing slots changing the resource description on-the-fly, depending on the workload.

In ServiceSs, the interaction with a PaaS resource is implemented in two parts: (i) a connector to acquire more computing slots (as in the IaaS case); (ii) a JavaGAT adaptor, to perform data transfer and job submission operations to the PaaS resource. Both the connector and the adaptor use the API of a specific vendor.

### *6.2.1 Microsoft Azure*

This subsection describes the Azure JavaGAT adaptor, which enriches ServiceSs with data management and

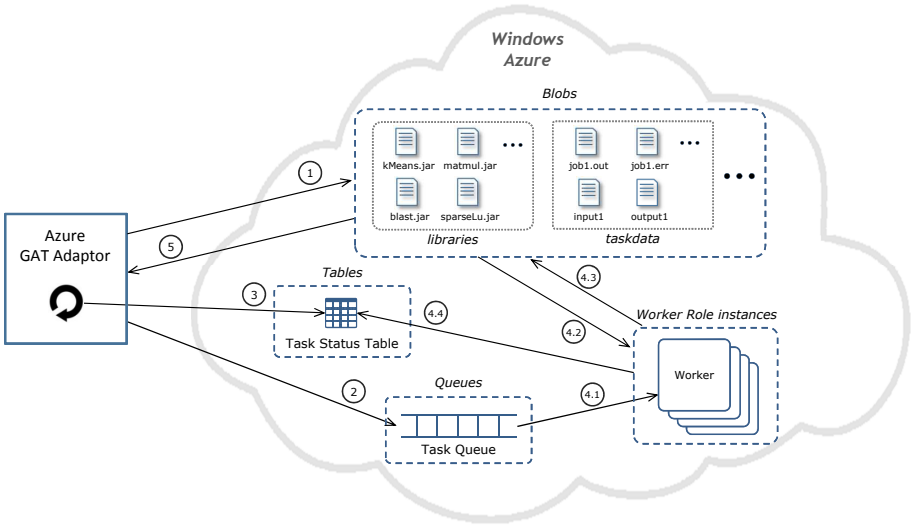



**Fig. 6** The Azure GAT adaptor architecture.

execution capabilities making it interoperable with the Azure platform.

The adaptor is composed of two subcomponents. The data management is supported by the Azure File Adaptor component, which allows to read and write data on the Azure Blob Storage (*Blobs*), deploy the needed *libraries* to execute on the platform, and store the jobs input/output data (*taskdata*). The second subcomponent, the Azure Resource Broker Adaptor, is responsible for the job submission. Following the Azure Work Queue pattern, this subcomponent adds into a *Task Queue* the ones that must be executed by a *Worker*.

Thus, in order to keep the runtime informed about each job execution, the status of the job is updated in a *Task Status Table*.

The numbered components in Figure 6 correspond to the items in the list below, which describes the different stages of a remote job execution. When the ServiceSs runtime submits a job $j$ into the platform it follows the next steps:

1. The Azure GAT adaptor, through the Azure File Adaptor, prepares the execution environment uploading the input application files and libraries into the Blob containers, *taskdata* and *libraries*.
2. The adaptor, via the Azure Resource Broker, inserts a job $j$ description into the *Task Queue*.
3. Then, it sets the status of the job $j$ to *Submitted* in the *Task Status Table* polling it periodically in order to monitor the status until it becomes *Done* or *Failed*.
4. An idle worker $W$ picks the job $j$ description from the queue, parses the parameters and runs the task. This step could be divided into the following substeps:
   - 4.1. The worker $W$ takes the job $j$ from the *Task Queue* starting its execution on a virtual resource, and the worker sets the status of $j$ to *Running*.
   - 4.2. The worker gets the needed input data and libraries according to the description of $j$. Then, it performs a file transfer from where the input data is located (Blob storage), to the local storage of the resource, launching, finally, the job.
   - 4.3. After job completion, the worker $W$ moves the resulting files in the *taskdata* Blob container.
   - 4.4. The worker updates the status of the job into the *Task Status Table* setting it to a final one that could be *Done* or *Failed*.
5. When the adaptor detects that the job $j$ execution has been completed, it notifies the end of the execution to the runtime which will look for new dependency-freed jobs. If the output files are not going to be used by any other job, the runtime downloads them from the Azure Blob Storage.

## 7 Evaluation

In order to evaluate the performance of ServiceSs, a set of experiments has been conducted using three different configurations: *i*) Cloud Federation: two clouds managed by the EMOTIVE Cloud and OpenNebula IaaS middleware; *ii*) IaaS Multi-Cloud: the combination of an EMOTIVE Cloud and Amazon EC2 VMs; *iii*) IaaS + PaaS Multi-Cloud: EMOTIVE Cloud plus Microsoft Azure instances.

The next subsection will describe the applications used in the tests, prior to the presentation of the results.

### 7.1 Applications

Two applications were chosen to run the experiments presented in this paper:

- *Gene Detection*: the tests in sections 7.2 and 7.3 executed a real example of an e-Science composite service, used for the detection of genes and programmed with ServiceSs. This service first finds a set of relevant regions (genes) in a DNA sequence for a given input protein, and then analyses only those genes. A more detailed description of Gene Detection can be found in [55].
- *Data Mining*: A data mining application has been adapted to run in a cloud environment with ServiceSs. Such application runs multiple instances of the same classification algorithm on a given dataset, obtaining multiple classification models, then it chooses the one which classifies in a more accurate way. A more detailed description of Data Mining can be found in [43].

Please note that these applications are written as sequential programs. However, they encompass invocations to different computations, either regular methods or external services, and some of these invocations can execute concurrently. This makes them suitable to be executed with ServiceSs, selecting such computations as ServiceSs CEs. In addition, in some cases the computations have data dependencies, which are dynamically discovered by ServiceSs to build the workflow of each application and exploit its inherent parallelism at execution time.

### 7.2 Cloud Federation

This section describes the evaluation of ServiceSs in a scenario composed by a cloud federation of two clouds, one managed by EMOTIVE Cloud and another one by OpenNebula. As said before, these tests have been performed by running a gene detection application.

#### *7.2.1 Testbed*

The structure of the used testbed is depicted in Figure 7 and is composed by the following elements:

- *Master*: laptop where the master runtime of ServiceSs is executed and the main code of the application is run. Also, an OpenVPN [17] client is used to be able to contact the virtual machines in the cloud federation.
- *Task server*: machine running an Apache Tomcat 7.0 WS [4] container, which hosts a service that offers the service CE operations. Such container is contacted by the ServiceSs runtime to execute service tasks called from the composite.
- *Cloud federation*: cluster managed by EMOTIVE Cloud and OpenNebula. The cluster has a front-end node that acts as an OpenVPN server, EMOTIVE scheduler and OpenNebula front-end. Besides, a total of 7 nodes are used for hosting VMs: 3 nodes with two eight-core AMD Opteron 6140 at 2.6 GHz processors, 32 GB of memory and 2 TB of storage each; 4 nodes with two six-core Intel Xeon X5650 at 2.67 GHz processors, 24 GB of memory and 2 TB of storage each. The nodes are interconnected by a Gigabit Ethernet network. In these tests, EMOTIVE manages 2 AMD and 4 Intel machines, while OpenNebula controls 1 AMD machine, all of them running XEN 4.0.1 hypervisor. The Master, Task server and the clouds are all located in the BSC premises in Barcelona, Spain.

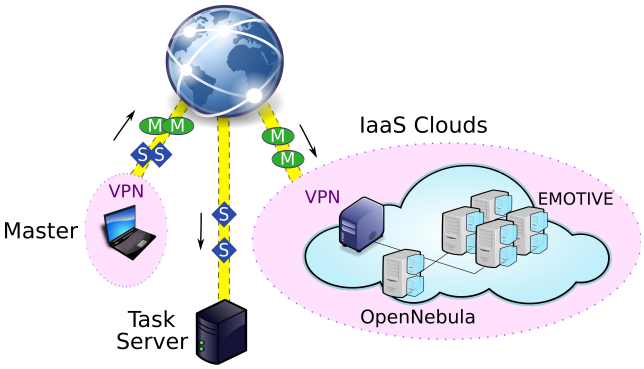



**Fig. 7** Testbed comprising two clouds: one managed by EMOTIVE Cloud and another one managed by OpenNebula. The GeneDetection application is executed in a laptop, which contacts the VMs of the clouds through a VPN. An external server publishes the operations corresponding to service CEs.

To launch an execution, ServiceSs runs the main code of the application in the master machine issuing Web Service requests to the Task server and creating VMs to execute methods in either EMOTIVE or OpenNebula. In this case, since both infrastructure providers offer an OCCI interface, ServiceSs can use the same connector to interact with them.

#### *7.2.2 Results*

In order to evaluate ServiceSs in the described testbed, a series of executions of the gene detection application have been performed with different number of VMs in each cloud. The VMs always consisted of 4 cores and 2 GB of memory. Figure 8 shows the execution times obtained for each configuration, the first four VMs (i.e. up to 16 cores) were created using only EMOTIVE in

the Intel nodes since their performance is slightly better, and for the executions with 32 and 48 cores, 4 VMs were created in OpenNebula and the rest in EMOTIVE to validate the interoperability.

As it can be seen in the plot, despite the execution time keeps on descending, the application does not scale as it would be expected. This is because many Web Service requests are issued at the same time and the task server becomes a bottleneck. However, using more cores always implies an increase in performance because the time spent in the computational part is reduced.

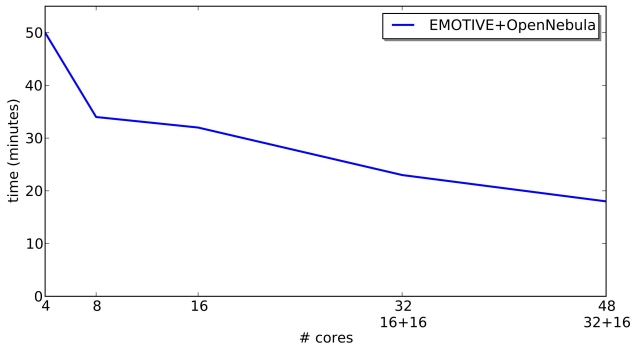



**Fig. 8** Execution times of the application when using different number of cores. The 32 and 48 cases were done with 16 cores in VMs under OpenNebula and the rest in EMOTIVE.

## 7.3 IaaS Multi-Cloud

This series of tests provides some performance results of ServiceSs when combining a cloud managed by EMOTIVE and the Amazon EC2 public cloud. Next we describe the testbed and the results obtained.

### *7.3.1 Testbed*

The testbed used in the experiments is formed by the following actors and infrastructures (see Figure 9):

- *Client*: Java application that invokes the gene detection composite service.
- *Composite server*: machine running an Apache Tomcat 7.0 WS container that hosts the gene detection service. It is a dual-core Intel Core i7 at 2.8 GHz, 8 GB of RAM and 120 GB of disk space. Both the composite's main program and the ServiceSs master runtime execute in this machine. This machine also runs an OpenVPN client.
- *Task server*: same as in section 7.2.
- *EMOTIVE Cloud*: the same cluster described in section 7.2. In these tests, the cluster was entirely managed by EMOTIVE Cloud. The Client, Composite server, Task server and EMOTIVE Cloud are all located in the BSC premises in Barcelona, Spain.
- *Amazon EC2*: public IaaS cloud provider. In the tests, all the Amazon VMs are deployed in the European Union West zone, which corresponds to a data center located near Dublin, Ireland.

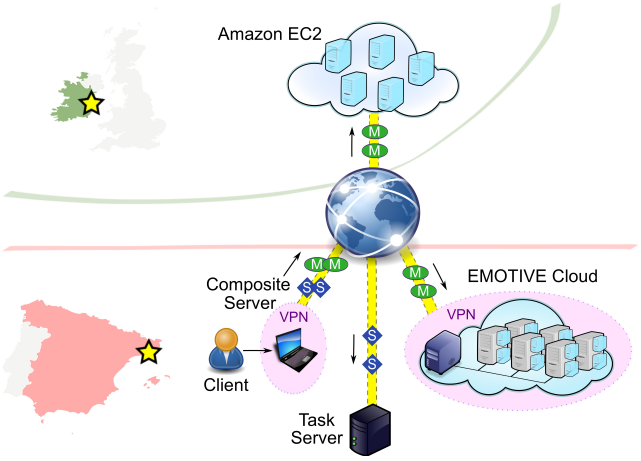



**Fig. 9** Testbed comprising two clouds: an EMOTIVE Cloud located at BSC and the Amazon EC2 public cloud (Ireland data center). The GeneDetection composite service is deployed in a server machine, which contacts the VMs of the EMOTIVE Cloud through a VPN. An external server publishes the operations corresponding to service CEs.

A typical execution begins when a Client issues a WS invocation request to the gene detection service published in the Composite server. This triggers the execution of the composite, leading to the creation of new method and service tasks. The ServiceSs runtime executes service tasks by issuing WS requests to the Task server container. Method tasks are run in VMs on the EMOTIVE Cloud or on Amazon EC2. In the case of the EMOTIVE Cloud, the Composite server and the VMs belong to the same virtual private network, so that they can communicate through SSH. Regarding Amazon, the VMs are also contacted by SSH to their public IP addresses. All the VMs run a Linux distribution where the ServiceSs worker runtime, BLAST and GeneWise have been pre-installed.

### *7.3.2 Results*

The tests in this subsection will focus in the most computationally-intensive part of the GeneDetection composite, that is, the execution of the GeneWise algorithm on a set of relevant regions of the genomic sequence. This part of the application generates a graph with the shape of a reversed binary tree, like the one in Figure 10, which first runs GeneWise on every relevant region previously found and then merges all the partial reports into one.

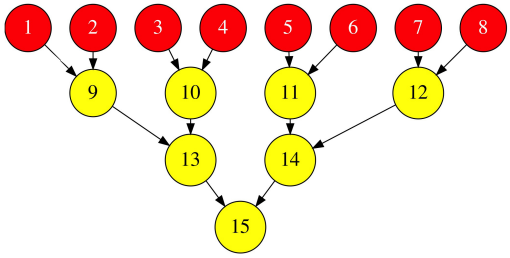



**Fig. 10** Graph generated by the GeneWise computation in the gene detection composite, for an execution that finds 8 relevant regions in the genomic sequence. Red (dark) tasks correspond to the *genewise* method, whereas yellow (light) ones represent calls to *mergeGenewise*.

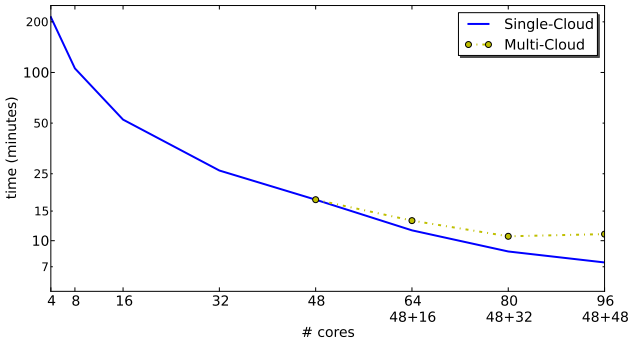



**Fig. 11** Execution times of the GeneWise computation, with EMOTIVE VMs only ('Single-Cloud') and a combination of EMOTIVE and public Amazon VMs ('Multi-Cloud').

In these experiments, the VMs were created beforehand, and so there is no delay associated with progressively acquiring/releasing VMs. The EMOTIVE VMs have 4 cores, 2 GB of RAM and 1 GB of storage (home directory). The Amazon VMs are of type 'm1.xlarge' (extra large), which also features 4 cores, 15 GB of RAM and 1690 GB of storage. On the other hand, the scheduling algorithm applied takes into account data locality when choosing the destination VM of a task, in order to reduce the number of transfers.

Figure 11 depicts the execution times (in logarithmic scale) of the GeneWise computation for different numbers of cores, more precisely the average of three executions per number of cores. In each execution, a total of 3068 *genewise* and 3067 *mergeGenewise* tasks are generated, with an average duration per task of 12 seconds and 200 milliseconds, respectively. One line corresponds to runs with only EMOTIVE VMs ('Single-Cloud'), while the other line plots the combination of both EMOTIVE and Amazon VMs ('Multi-Cloud').

The measures show that ServiceSs achieves good scalability, especially when running the whole computation in one cloud provider ('Single-Cloud'). In the 'Multi-Cloud' executions, the results are affected by the distributed nature of the testbed (Figure 9). When distributing the tasks of the GeneWise computation graph over more than one provider, task dependencies eventually lead to data transfers between VMs in different providers, even if the locality-aware scheduling algorithm of ServiceSs tries to minimize the number of transfers. Providers can be geographically dispersed, like in our testbed, and consequently latencies can be higher than in other cloud scenarios. Moreover, there can be no connectivity between VMs of different providers: this happens in our case, where every data transfer between an EMOTIVE and an Amazon VM passes through the Composite server first. However, the fact that the GeneWise reports are rather small (up to a few kilobytes) helps scale better the application even under those conditions. Please note that the difference in performance when using a single cloud or multiple clouds is not due to the overhead of the runtime, but to the fact of distributing data and computations over distant cloud sites.

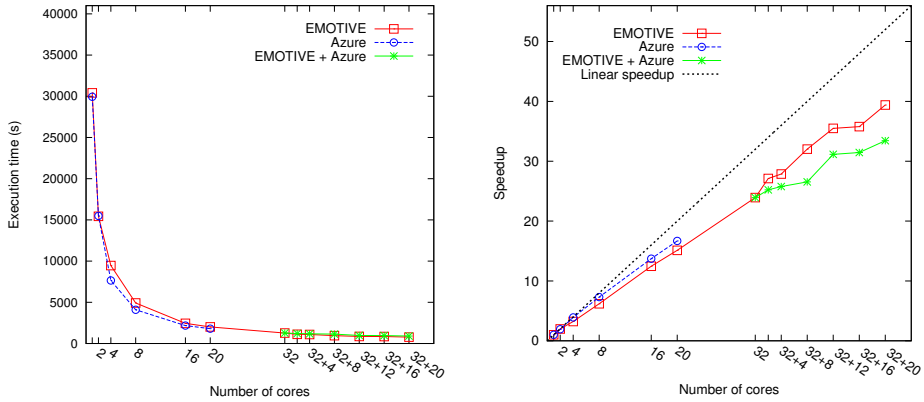
On the other hand, task granularity also plays an important role in the performance of GeneWise. The duration of the *genewise* is quite varying, which challenges the load balancing mechanism of the runtime, while the small execution time of *mergeGenewise* complicates the overlapping of transfers and computation, particularly for the 'Multi-Cloud' case where the average transfer time is higher. Nevertheless, even if they are hardly worth to distribute, the *mergeGenewise* method invocations are run as tasks to prevent the main program from having to reduce all the partial GeneWise reports.

## 7.4 IaaS + PaaS Multi-Cloud

This section describes the evaluation of ServiceSs when using EMOTIVE Cloud and Microsoft Azure platform under a data mining application environment. The final goal is to compare the behavior and performance of our framework in single-cloud and multi-cloud scenarios.

### *7.4.1 Testbed*

On this experiment, a testbed with 14 quad-core virtual machines with 5 GB of memory, 2 GB of disk space and running Linux distribution has been created on EMOTIVE Cloud. In the same way, the public infrastructure, based on Windows Azure, was composed by up to 20 small virtual appliances equivalent to 1.6 GHz single core processor, 1.75 GB of memory and 225 GB of disk space. In order to reduce the impact of data transfer on the execution time, the Azure's Affinity Group feature has been exploited allowing the storage and computing




(a) (b)

**Fig. 12** Execution time and speedup values depending on the number of cores.

servers to be located in the same data center improving the global performance.

On execution, the *covertype* [21] dataset has been used as data source. This dataset contains information about forest cover type of a large number of sites in the United States. Each instance corresponds to a site observation and contains 54 attributes that describe the main features of a site (e.g., elevation, aspect, slope, etc.). The original dataset is made of 581.012 instances sizing 72 MB. A subset of it, with 290.000 instances, has been used creating a new 36 MB one.

For evaluating the performance of the application, a set of experiments has been conducted using three different configurations: i) execution only on EMOTIVE Cloud using up to 52 cores; ii) execution only on Azure using up to 20 cores; iii) a Multi-Cloud configuration using both (32+20 cores).

### *7.4.2 Results*

As depicted in Figure 12, execution times and speedup from 1 to 20 cores are similar in both cases when a single provider (EMOTIVE or Azure) is used, keeping a quasi-linear speedup along the execution up to the point where cloud bursting starts. At this time, two different experiments has been conducted. Starting from 32 EMOTIVE cores, 20 Azure and EMOTIVE cores have been added progressively in order to evaluate the system behaviour when running on this two different environments: 20 locally-wired Cores vs 20 Internet-wired ones. In order to calculate network speed differences a 753 MB file has been transferred across different local VMs and across Internet from local to Azure VMs obtaining a 752.9 Mbps vs 13.2 Mbps speeds respectively. Although the application is not data-intensive, the scenario condition and a workload unbalance, due to the impossibility to adjust the total number of tasks (constrained by the specific use case) to the amount of available resources, influences on the overall application performance showing a slightly variable curve.

Despite this, when the number of resources allows a good load balancing, the speedup curve recovers some of the performance loss as depicted in 32+16 case where the gain starts to increase again. Thus, in general cases, when the workload does not depend on the application input, the ServiceSs runtime scheduler is able to adapt the number of tasks to the number of available resources.

## 7.5 Runtime Overhead Analysis

The main goal of this section is to evaluate the ServiceSs runtime overhead running a synthetic benchmark application on a cloud scenario (defined on section 7.2), determining the performance loss within the runtime system according to the amount of tasks and computing resources.

### *7.5.1 Runtime benchmarking*

The presented experiment varies the amount of tasks keeping constant the volume of transferred data and execution time of each task.

In order to measure the overhead of the runtime, four aspects have been considered. The *Dependency Management* overhead has been quantified considering the time spent on analyzing tasks data accesses and the time needed to add and remove them from the dependency graph. The *Scheduling* time has been measured considering the time since a task becomes dependency-free until its assignment to a certain resource as well as the time to free the resource once the task finishes. The *Job Submission* time is defined from the instant when the decision to execute a task is made until this

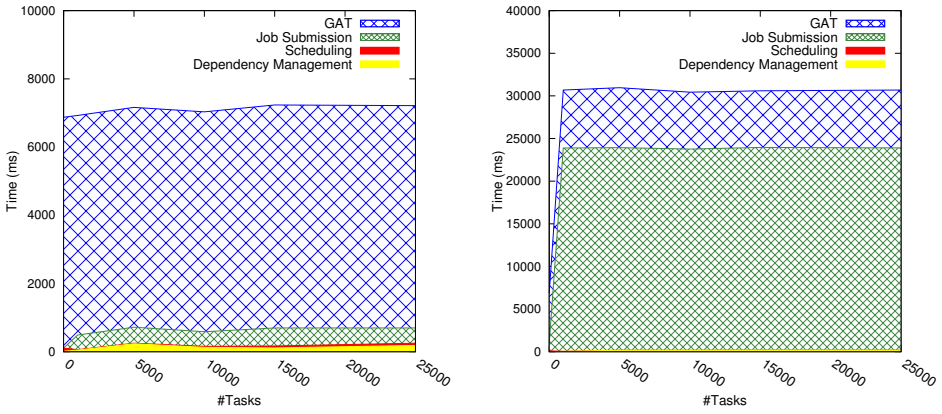



**Fig. 13** ServiceSs runtime cumulative per-component overhead, depending on the amount of tasks and resources. Figure 13(a) and Figure 13(b) represent the results running on 16 and 96 computing slots respectively.

decision is communicated to JavaGAT. The *GAT* overhead considers the time spent by JavaGAT to submit and finalize the task.

### *7.5.2 Results*

The tests conducted in this section aim to stress the runtime system, forcing it to handle up to 25000 tasks in two configurations with 16 and 96 computing nodes.

Figure 13(a) highlights that the general overhead does not depend on the number of managed tasks and that it becomes almost constant during the set of experiments. It is worth noting that the use of JavaGAT as communication layer introduces around 90% of the total overhead.

In the other experiment, whose results are depicted in Figure 13(b), it can be observed that while the overhead is constant and does not depend on the number of tasks, the *Job Submission* time grows again due to the JavaGAT component. Although this component allows the execution of many tasks in a concurrent way, it can only manage one job submission request at a time, with an average processing time of 300 ms. This time is accumulated when several tasks are ready for submission causing an overhead of about 60%.

## 8 Related Work

We have classified the related work into four categories: PaaS frameworks, programming models for computing intensive workloads, workflows managers and programming models for big data. For each one of the considered categories we concentrate on the programmability features and on the interoperability capabilities offered.

### 8.1 PaaS Frameworks

Numerous Platform-as-a-Service (PaaS) solutions have appeared to facilitate the process of developing, deploying and running applications in the cloud. Some of them propose programming models that offer APIs to write applications. In the Microsoft Azure Cloud programming model the applications are structured in roles, which use APIs to communicate (queues) and to access persistent storage (blobs and tables). Microsoft Generic Worker [52] proposes a mechanism to develop a Worker Role that eases the porting of legacy code in the Azure platform. Even if the user does not have to change the core of the code, the creation of workflows is not automated, as in ServiceSs, but has to be explicitly enacted through separated executions. Google App Engine provides libraries to invoke external services and queue units of work (tasks) for execution; furthermore, it allows to run applications programmed in the MapReduce model. Contrarily to these platforms, ServiceSs does not require including any API call in the application code; CE creation (either from regular methods or services), data transfer and synchronization are handled automatically by the runtime. Moreover, data dependencies between CEs do not need to be managed manually in the application code, since they are resolved by the runtime. In Google App Engine the developer should take care of programming the orchestration. Finally, the aforementioned PaaS proposed by Microsoft and Google restrict the deployment and execution of their applications to their own infrastructure; oppositely, ServiceSs applications can run on top of any supported cloud provider. The CONTRAIL project is developing ConPaaS [50], a runtime environment for elastically hosting and deploying applications on several clouds as Amazon EC2 and OpenNebula. The ser-

vices are offered through abstractions that hide the real implementations and can be instantiated on multiple clouds. Applications in ConPaaS are composed of any number of these services programmed through a common generic Python interface for the service management and a Javascript part to extend the front-end GUI with service-specific information and control. The main difference with ServiceSs is the existence, in ConPaaS, of predefined patterns for the implementation of a service, thus forcing the users to write new code or to adapt the existing one to use the currently provided services. Currently only OpenNebula and Amazon EC2 cloud backends are supported.
In the mOSAIC project [44], a reference API has been designed which is composed of a set of layered APIs focusing on achieving interoperability between clouds. The developer has to provide applications requirements through the API while the runtime search for cloud services and offerings matching the requirements. However, a strong requirement is that end users are forced to adapt their applications to this newly created API that is interoperable with Amazon EC2, Eucalyptus [48], FlexiScale [7], CloudStack [3], and OpenNebula. Aneka is a .NET-based application development PaaS, which offers a runtime environment and a set of APIs that enable developers to build customized applications by using multiple programming models on public clouds as Azure, Amazon EC2, and GoGrid [9]. While the most important feature in Aneka is related to its runtime and how it manages dynamic provisioning and accounting of virtual resources, the programming framework it offers do not address the easy development of composite services and is limited to applications running on Windows machines.

### 8.2 Environments for computing intensive applications

In the computing intensive group we first consider frameworks that allow the programming and execution of high throughput applications; in this area the so called bag of tasks execution model is used to run applications composed of independent parallel tasks, often also referred to as embarrassingly parallel. The Ibis framework [25] implements BaTS [49] for the execution of bag of tasks loads in the cloud under budget control. Ibis, similarly to ServiceSs, is based on JavaGAT to access different middlewares and provides a range of programming models, from low-level message passing to high-level divide-and-conquer parallelism, using a common communication and resource management library. Differently from ServiceSs, the user has to explicitly implement the API corresponding to the specific pattern to port the application and to compile the code using specific scripts. The actual support to cloud is limited to Amazon EC2 through a Java GAT adaptor; a more extended interoperability is expected to be achieved by the adoption of SAGA API, in place of JavaGAT, in the near future. Swift [56] is a scripting language oriented to scientific computing which can automatically parallelize the execution of scripts and distribute tasks to various resources. The programming model is based on the specification of functions executing external programs and the input/output data associated with each execution. The data dependencies are resolved by the engine at execution time exploiting the implicit parallelism of the code in a similar way than ServiceSs. The execution in cloud is delegated to the Coaster System [34] that requires a manual provisioning of virtual machines before the execution and no elasticity features are provided. SAGA BigJob [41] is a pilot job system implemented as a SAGA adaptor and on top of SAGA adaptors. The Cloud BigJob attempts to hide from the user most of the differences between providers. However, the user is still required to perform some manual steps, such as setting up the VM image and keys, prior to using the Cloud BigJob. Currently, Amazon EC2, Eucalyptus and Nimbus Clouds [28] are supported but using command line clients and not the specific provider's API. ProActive offers a new resource manager that has been developed in order to mix Cloud and Grid resources [23], but the programming model lacks proper service-orientation: although an active object can be deployed as a service, there is no special support for orchestration of several service active objects. The deployment phase implies the user intervention to configure the resources to be used and the installation of a scheduler and resource manager on the hosting provider.

### 8.3 Workflow Managers

Another kind of approach allows the graphical composition of an application workflow whose nodes can be services. Some vendors have implemented their own WS-BPEL [15] visual editor to create orchestrations of services which, at their turn, can also be published as services; BPEL features control flow statements (if-then-else, while) and the data flow is defined by linking services. Taverna [46] is a workflow language and computational model designed to support the automation of complex, service-based and data-intensive processes. Taverna supports the detection of tasks free of data-dependences and is able to execute them in parallel. A prototype version has been tested on a private research cloud deploying Taverna Servers on a set of virtual machines but neither plugins are offered to execute

tasks on public clouds nor the possibility exists of elastically modify the pool of resources. Pegasus [26] is a workflow manager that automatically maps high-level workflow descriptions, provided as XML files, onto distributed resources as clouds. The execution in cloud is not straightforward because forces the user to manually pre-deploy several nodes configured as Condor workers, opposite to ServiceSs that automatically deploys the applications on dynamically created virtual machines. In contrast to these approaches, the workflow of a ServiceSs application (the CE dependency graph) is not defined graphically, but dynamically created as the main program runs: each invocation of a method or service is replaced on-the-fly by the creation of an asynchronous CE which is added to the graph. ServiceSs only requires skills in sequential programming; no knowledge in multithreading, parallel/distributed programming or service invocation is necessary. While semantic information in CEs is not supported yet, it could be added as an interface annotation instead of selecting a particular instance of a service. JOLIE [47] allows to program textually service compositions, but unlike ServiceSs it uses a custom syntax and requires the user to deal with parallelism and synchronization explicitly.

### 8.4 Environments for data intensive applications

In the last group we consider those programming models and frameworks related to the processing and generation of large data sets. In this area MapReduce programming model is a widely used and implemented model that provides good performance on clouds architectures above all on data analytics applications on large data collections. In particular, we consider two variants of MapReduce that are relevant in clouds. In one case there is no real reduce step and instead just the map operation is applied to each input in parallel. This is often called an embarrassingly parallel computation. At the other extreme there are computations that use MapReduce inside an iterative loop. Microsoft Daytona [19] presents an iterative MapReduce runtime for Windows Azure designed to support a wide class of data analytics and machine learning algorithms. Twister [33] is an enhanced MapReduce runtime with an extended programming model that supports iterative MapReduce computations efficiently. A tailored version allows the programming of MapReduce applications on Azure. In both cases the implementation forces the user to submit the code in the Azure cloud and there is no elastic provision of VMs. Hadoop [27] is the most popular open source implementation of MapReduce on top of the Hadoop Distributed File System (HDFS) [27]. The use of Hadoop avoids the lock in to a specific platform allowing to execute the same MapReduce application on any Hadoop compliant service, as the Amazon Elastic MapReduce [2], but this still requires manual intervention of the user to configure the resources. In [45] a framework is presented to run MapReduce applications on hybrid clouds managed by Aneka. Regarding the expressiveness of the models, ServiceSs is more flexible than MapReduce because its applications can generate any arbitrary CE (task) graph. Such generality is also pursued by Dryad [36], but in this case the graph has to be created programmatically by means of C++ libraries and overloaded operators.

## 9 Conclusions and Future Work

This work presented the features of ServiceSs, a framework for the development, deployment and execution of parallel applications, business and scientific workflows and compositions of services mixing services and code on distributed infrastructures. ServiceSs provides users with a simple sequential programming model that does not require the use of APIs and enables the execution of the same code on different cloud providers. The ServiceSs runtime is designed to provide interoperability with different IaaS and PaaS offerings through the implementation of connectors, enabling the developed services to run on hybrid deployments. The existing connectors implement open community specifications such as OCCI for virtual machine management, or commercial offerings such as EC2 and Azure. These interoperability features have been evaluated through the execution of several use cases on hybrid testbeds, demonstrating that the runtime is able to elastically schedule the tasks of the services dynamically by distributing the load across resources from multiple providers.

The ServiceSs programming model, runtime and IDE have been mainly developed during the OPTIMIS [29] project. Besides, it has been adopted for the porting of scientific applications in several infrastructures such as VENUS-C [22] and EUBrazilOpenBio [39], and is now leveraged in European Grid Infrastructure (EGI) [6] as enabling technology for the execution of composed workflows on the federated cloud testbed.

Future work includes the extension of the OCCI connector to implement the full specification in order to achieve interoperability with other clouds as OpenStack, and the extension of the Azure adaptor to support the dynamic provisioning of instances.

In terms of the runtime, the final year of the OPTIMIS project will evolve it in order to achieve control of all developed services from the same Service Provider, enabling to do further research in specifying different policies at SP level on how resources will be managed

for running the full set of SP services. We will also further investigate to include the support of nested Core Elements, which allows the main code of the application to invoke CEs composed by finer grain CEs (achieving hierarchical compositions of CEs). Other efforts chase to achieve the auto-scaling of machines to minimize economic costs, in order to meet established application deadlines by forecasting execution times of CEs.

At the light of the evaluation results, in order to improve the scenario with multiple Clouds, ServiceSs could link entire composite runs to specific providers. In a server that receives multiple requests for composites, this could be done by bursting to a public cloud the whole execution of a composite, instead of offloading some of the tasks of a composite that is already being executed in VMs hosted on-premises. This strategy would execute full graphs in the VMs of a single provider causing data transfers to always happen inside the boundaries of a single cloud thus avoiding expensive inter-provider transfers.

**Acknowledgements** This work has been supported by the Spanish Ministry of Science and Innovation (contracts TIN2012-34557, TIN2007-60625, CSD2007-00050 and CAC2007-00052), by Generalitat de Catalunya (contract 2009-SGR-980), by the European Commission (OPTIMIS project, Grant Agreement Number: 257115, EUBrazilOpenBio project, Grant agreement no. 288754) and by the grant SEV-2011-00067 of Severo Ochoa Program, awarded by the Spanish Government.

## References

1. Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/. Last visited on 4/16/2013.
2. Amazon elastic map reduce. http://aws.amazon.com/documentation/elasticmapreduce/. Last visited on 8/31/2013.
3. Apache cloudstack. http://cloudstack.apache.org, Last visited on 8/31/2013.
4. Apache Tomcat. http://tomcat.apache.org/. Last visited on 8/31/2013.
5. Eclipse: The eclipse foundation open source community website. http://www.eclipse.org/, Last visited on 8/31/2013.
6. Egi-inspire white paper. http://go.egi.eu/pdnon. Last visited on 4/16/2013.
7. Flexiscale. http://www.flexiscale.com, Last visited on 8/31/2013.
8. Foster I et. al. OGSA Basic Execution Service Version 1.0. Grid Forum Document GFD-RP. 108. 11/13/2008. http://www.ogf.org/documents/GFD.108.pdf. Last visited on 8/31/2013.
9. Gogrid cloud hosting. http://www.gogrid.com.
10. Google App Engine. http://code.google.com/appengine.
11. Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.
12. Java programming assistant (javassist. http://www.javassist.org. Last visited on 8/31/2013.
13. JavaGAT. http://www.cs.vu.nl/ibis/javagat.html. Last visited on 8/31/2013.
14. Microsoft Azure. http://www.microsoft.com/azure.
15. OASIS Web Services Business Process Execution Language. http://www.oasis-open.org/committees/wsbpel/.
16. Open Cloud Computing Interface. http://occi-wg.org, Last visited on 4/16/2013.
17. Open VPN. http://openvpn.net/. Last visited on 8/31/2013.
18. Ovf4one. http://occi-wg.org/2012/05/03/occi-and-ovf/. Last visited on 4/16/2013.
19. Project daytona. http://research.microsoft.com/en-us/projects/daytona. Last visited on 4/16/2013.
20. The HPC Basic profile specification. http://www.ogf.org/documents/GFD.114.pdf. Last visited on 4/16/2013.
21. Us forest service (usfs) cover type. http://kdd.ics.uci.edu/databases/covertype/covertype.html. Last visited on 4/16/2013.
22. Virtual multidisciplinary ENvironments USing Cloud infrastructures Project. http://www.venus-c.eu. Last visited on 8/31/2013.
23. B. Amedro, F. Baude, D. Caromel, C. Delbe, I. Filali, F. Huet, E. Mathias, and O. Smirnov. An efficient framework for running applications on clusters, grids and clouds. In *Cloud Computing: Principales, Systems and Applications*. Springer Verlag, 2010.
24. A. Anjomshoaa, F. Brisard, M. Drescher, D. Fellows, A. Ly, S. Mcgough, D. Pulsipher, and A. Savva. Job Submission Description Language (JSDL) Specification, Version 1.0. Technical report, Global Grid Forum, June 2005.
25. H. Bal, J. Maassen, R. van Nieuwpoort, N. Drost, R. Kemp, T. van Kessel, N. Palmer, G. Wrzesińska, T. Kielmann, K. van Reeuwijk, F. Seinstra, C. Jacobs, and K. Verstoep. Real-world distributed computer with ibis. *IEEE Computer*, 43(8):54–62, 2010.
26. G. B. Berriman, E. Deelman, G. Juve, M. Rynge, and J.-S. Vöckler. The application of cloud computing to scientific workflows: a study of cost and performance. *Physical and Engineering Sciences*, 371(1983), Jan. 2013.
27. D. Borthakur. *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.
28. J. Bresnahan, K. Keahey, and M. Wilde. Nimbus platform: Managing deployments in multi-cloud environments. *Science Cloud Summer School 2012*, 07/2012 2012.
29. A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forgó, T. Sharif, and C. Sheridan. OPTIMIS: A holistic approach to cloud service provisioning. *Future Generation Computer Systems*, 28(1):66 – 77, 2012.
30. J. Giménez, J. Labarta, F. X. Pegenaute, H.-F. Wen, D. Klepacki, I.-H. Chung, G. Cong, F. Voigtländer, and B. Mohr. Guided performance analysis combining profile and trace tools. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 513–521, Berlin, Heidelberg, 2011. Springer-Verlag.
31. Í. Goiri, J. Guitart, and J. Torres. Elastic Management of Tasks in Virtualized Environments. In *Proceedings of the XX Jornadas de Paralelismo (JP 2009), A Coruña, Spain, September 16–18*, pages 671–676, 2009.
32. T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid applications,

High-Level Application Programming on the Grid. *Computational Methods in Science and Technology (CMST)*, 12(1):7–20, 2006.
33. T. Gunarathne, B. Zhang, T.-L. Wu, and J. Qiu. Scalable parallel computing on clouds using twister4azure iterative mapreduce. *Future Generation Computer Systems*, 29(4):1035 – 1048, 2013. ¡ce:title¿Special Section: Utility and Cloud Computing¡/ce:title¿.
34. M. Hategan, J. Wozniak, and K. Maheshwari. Coasters: Uniform resource provisioning and access for clouds and grids. In *Proceedings of the 2011 Fourth IEEE International Conference on Utility and Cloud Computing*, UCC '11, pages 114–121, Washington, DC, USA, 2011. IEEE Computer Society.
35. J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2002.
36. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.
37. D. Lezzi, S. Memon, R. Rafanell, H. Soncu, M. Riedel, and R. M. Badia. Interoperable execution of escience applications on grids & clouds through open standards. In *Proceedings of the Unicore Summit 2012*, IAS. Forschungszentrum Jülich GmbH, 2012.
38. D. Lezzi, R. Rafanell, A. Carrión, I. B. Espert, V. Hernández, and R. M. Badia. Enabling e-science applications on the cloud with compss. In *Proceedings of the 2011 international conference on Parallel Processing*, Euro-Par'11, pages 25–34, Berlin, Heidelberg, 2012. Springer-Verlag.
39. D. Lezzi, R. Rafanell, E. Torser, R. De Giovanni, I. Blanquer, and R. M. Badia. Programming ecological niche modeling workflows in the cloud. In *The 27th IEEE International Conference on Advanced Information Networking and Applications (AINA-2013)*, BARCELONA, Spain, Mar. 2013.
40. I. Livenson and E. Laure. Towards transparent integration of heterogeneous cloud storage platforms. In *Proceedings of the fourth international workshop on Data-intensive distributed computing*, DIDC '11, pages 27–34, New York, NY, USA, 2011. ACM.
41. A. Luckow, L. Lacinski, and S. Jha. Saga bigjob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 135–144, 2010.
42. R. Mach, R. Lepro-Metz, S. Jackson, and L. McGinnis. Usage record format recommendation gfd-rp. 098. In *Open Grid Forum Recommendation*, 2007.
43. F. Marozzo, F. Lordan, R. Rafanell, D. Lezzi, D. Talia, and R. M. Badia. Enabling cloud interoperability with compss. In C. Kaklamanis, T. S. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par*, volume 7484 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2012.
44. B. Martino, D. Petcu, R. Cossu, P. Goncalves, T. Mhr, and M. Loichate. Building a mosaic of clouds. In M. Guarracino, F. Vivien, J. Trff, M. Cannatoro, M. Danelutto, A. Hast, F. Perla, A. Knpfer, B. Martino, and M. Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 571–578. Springer Berlin Heidelberg, 2011.
45. M. Mattess, R. N. Calheiros, and R. Buyya. Scaling mapreduce applications across hybrid clouds to meet soft deadlines. In *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA), 2013, Barcelona, Spain*, 2013.
46. P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna, reloaded. In M. Gertz, T. Hey, and B. Ludaescher, editors, *SSDBM 2010*, Heidelberg, Germany, June 2010.
47. F. Montesi, C. Guidi, and G. Zavattaro. Composing services with jolie. In *Web Services, 2007. ECOWS '07. Fifth European Conference on*, pages 13 –22, nov. 2007.
48. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.
49. A.-M. Oprescu, T. Kielmann, and H. Leahu. Budget estimation and control for bag-of-tasks scheduling in clouds. *Parallel Processing Letters (PPL)*, 21(2):219–243, 2011.
50. G. Pierre and C. Stratan. ConPaaS: a platform for hosting elastic cloud applications. *Internet Computing, IEEE*, 16(5):88–92, September-October 2012.
51. O. Sefraoui, M. Aissaoui, and M. Eleuldj. Article: Openstack: Toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, October 2012. Published by Foundation of Computer Science, New York, USA.
52. Y. Simmhan, C. van Ingen, G. Subramanian, and J. Li. Bridging the gap between desktop and the cloud for escience applications. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, pages 474–481, Washington, DC, USA, 2010. IEEE Computer Society.
53. B. Sotomayor, R. Montero, I. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *Internet Computing, IEEE*, 13(5):14 –22, sept.-oct. 2009.
54. E. Tejedor and R. Badia. Comp superscalar: Bringing grid superscalar and gcm together. In *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*, pages 185–193. IEEE, 2008.
55. E. Tejedor, J. Ejarque, F. Lordan, R. Rafanell, J. Álvarez, D. Lezzi, R. Sirvent, and R. M. Badia. A Cloud-unaware Programming Model for Easy Development of Composite Services. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science*, CloudCom '11, Athens, Greece, November 2011.
56. M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Comput.*, 37(9):633–652, Sept. 2011.