Future Generation Computer Systems 29 (2013) 1417–1430

Contents lists available at SciVerse ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Portable Cloud applications—From theory to practice☆

Dana Petcu [a,b,*], Georgiana Macariu [a,c], Silviu Panica [a,b], Ciprian Crăciun [a,b]

[a] Institute e-Austria Timişoara, Romania
[b] West University of Timişoara, Romania
[c] Politehnica University of Timişoara, Romania

ARTICLE INFO

Article history:
Received 16 December 2011
Accepted 18 January 2012
Available online 27 January 2012

Keywords:
Cloud computing
Open-source API implementation
Application portability

ABSTRACT

The adoption of the Cloud computing concept and its market development are nowadays hindered by the problem of application, data and service portability between Clouds. Open application programming interfaces, standards and protocols, as well as their early integration in the software stack of the new technological offers, are the key elements towards a widely accepted solution and the basic requirements for the further development of Cloud applications.

An approach for a new set of APIs for Cloud application development is discussed in this paper from the point of view of portability. The first available, proof-of-the-concept, prototype implementation of the proposed API is integrated in a new open-source deployable Cloudware, namely mOSAIC, designed to deal with multiple Cloud usage scenarios and providing further solutions for portability beyond the API.

1. Introduction

The Cloud computing paradigm promises to make the infrastructure programmable. Moreover it has the potential to address the programmability of resources in general, either infrastructure or software ones. However, the adoption of this vision is still hindered nowadays by the lack of proper technology, knowledge and trust. One of the main problems spanning over all these three reasons of concerns is the low level of portability of applications and services based on Clouds. The vendor lock-in with respect to both low-level resource management and application-level services is related also to the lack of world-wide adopted standards or interfaces to leverage the dynamic landscape of Cloud related offers. Since the Cloud service providers are increasing in number every day and they are bringing new but proprietary interfaces, the problem becomes more acute with time passing, and the global agreement on an acceptable solution harder to be achieved.

The current trend of using services from several Clouds puts also a considerable pressure to overcome fast the portability problem using available technical solutions. The portability in a large market of Cloud offers should allow the consumers to be able to use services across different Clouds by seamless switching between providers or on-demand scaling out on external resources other than the daily ones.

A short analysis of the current approaches to portability issues, as the one provided in the next section, reveals the urgent need for a general agreement on the application programming interfaces (APIs), matching the specific requirements of Cloud computing, as well as for integrating existing open approaches in real implementations. However, application portability between Clouds means more than API portability (as we will discuss in the next section). Here we point towards a single scenario. An application may use software resources, for example message queues, which are provided only by very few vendors. Beside the resource programmability issue, this scenario brings another limitation because moving to another Cloud vendor is impossible if the vendor does not support the required resource.

Furthermore, the resource programmability is understood differently at infrastructure, platform or software as a service levels. In the case of Infrastructure as a Service (IaaS), the Cloud service consumer has a stronger control on the resources than in the case of grid computing, for example, but usually needs to follow a predefined API of the resource provider. Fortunately, following the needs of interoperability in the case of Cloud federations, there are considerable efforts dedicated to establish some standards in what concerns the APIs (at IaaS level), like OCCI or CDMI (see the list from the Appendix for references to the cited technologies). In the case of Platform as a Service (PaaS), the Cloud service consumers do not have anymore the strong control on the resources, the APIs being limited. Moreover the current APIs are imposing certain programming paradigms or programming

☆ This work was supported by the grant of the European CommissionFP7-ICT-2009-5-256910 (mOSAIC) and Romanian National Authority for Scientific Research, CNCS UEFISCDI, PN-II-ID-PCE-2011-3-0260 (AMICAS).

* Corresponding author at: Institute e-Austria Timişoara, Romania. Tel.: +40 256 592370; fax: +40 256 244834.

E-mail addresses: petcu@info.uvt.ro, dana@ieat.ro (D. Petcu), georgiana@ieat.ro (G. Macariu), silviu@info.uvt.ro (S. Panica), ccraciun@info.uvt.ro (C. Crăciun).

0167-739X/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2012.01.009

languages (for different practical reasons, e.g. to exploit the pre-existing technologies of the Cloud providers, or to simplify the migration towards Cloud of a certain category of applications). The diversity in this case cannot be overcome by standards and the alternative is to develop supplementary level of abstractions. Furthermore, the consumer can be unsatisfied by the reduced level of control, e.g. in terms of elasticity mechanism imposed by the provider.

In the above two cases the resource is an abstract representation of physical devices or parts of them. However, the resources can be also software components. Unfortunately there are few technologies dealing with the management of software resources in Cloud environments. In this context we should underline that there are two types of delivery models of Cloud software: deployable or hosted. The hosted software is appropriate for public Clouds and force the customer to comply to it already at the design phase of the application. The deployable software is more appropriate for private Clouds and give the freedom to create a mixture of technologies that can support a certain type of application. Each deployable software can be treated as a component of the application using it in a Cloud environment. If we introduce an abstract level in which these components are foreseen as programmable resources too (as sustained in this paper), a new dimension is added to the programmability in Clouds.

Cloud computing differentiates itself from other distributed computing paradigms through its apparent infinite elasticity. Elasticity is currently understood as the ability to on-demand scale-up and down the number of Cloud resources allocated for an application. While the elasticity has a deep implication on the management of the resources at the provider side, what finally counts for the owner of an application is that the resource usage pattern follows closely the one imposed by the application at runtime. The application itself should allow elasticity and therefore means to express the elasticity are needed. Web applications are matching very well the elasticity requirements and therefore are the most successful use cases of Clouds. However, the adoption of Clouds on large scale is hindered by the fact that legacy codes need to be re-written in order to take advantage of elasticity. In consequence, Cloud computing has been fast adopted by start-ups developing from scratch new applications and less by large corporations owning mission critical systems. Without the pressure of the large corporations, the non-portability of the applications developed for one Cloud to another Cloud is currently exploited by Cloud service providers for they own benefits (the vendor lock-in mentioned above).

Another issue hindering the Cloud adoption is the high investment in the development phase of a Cloud application. The current IaaS services still require a certain level of expertise in setting the remote environment for running the application. The current PaaSs eliminate this need but introduce already in the testing phase another need, that of using the Cloud resources of the PaaS provider (leading to high testing costs and again vendor lock-in). We consider that in order to improve the productivity of the Cloud application development we need Cloud environment simulators that allow the development of Cloud application at the developer site or desktop and the seamless porting of the tested application on remote resources. Such support for desktop development of Cloud application is practically not existing at this moment.

Summarizing the above described ideas, we consider in what follows that:

- the design of the Cloud related APIs is critical for the Cloud application portability;
- the software resources can be programmable in Cloud;
- the Cloud elasticity can be exploited at the application level;
- the usage of Cloud aware deployable software as resources can ensure the portability of Cloud application;
- the availability of a desktop simulator of Cloud environment can increase the productivity in the development of Cloud applications.

Following these ideas, the multi-national team of mOSAIC project (acronym for Open source API and Platform for Multiple Clouds) proposed a vendor-agnostic and language-independent set of open APIs supporting portability between Clouds, as well as a prototype deployable and portable open-source platform intended to compete with the current vendor-dependent and hosted PaaSs.

The aim of this paper is to present the mOSAIC's proposal and implementation benefits from the portability point of view, focusing mainly on its API proposal.

The novel aspects introduced in this paper are the following:

- description of the concepts behind mOSAIC API and its positioning versus other similar initiatives;
- analysis of the current approaches for portability of applications between Clouds;
- exemplify the portability approaches through mOSAIC's approach;
- considerations on the architecture of Cloud applications;
- experimental results highlighting the benefits of adopting a new architectural style for the Cloud applications.

2. Tackling with portability in the Cloud

Portability, in general, is the ability to use components or systems lying on multiple hardware or software environments. The portability between Clouds, or shortly Cloud portability, ensures that a targeted application, data or service will work in the same manner, regardless of the underlying Cloud services, and has a common method of programmatic interaction with these services.

The following section motivates the urgent need to solve the portability issues and presents the approaches that were undertaken until now, as well as the positioning of mOSAIC proposal in the solutions' landscape.

2.1. When the need of portability occurs

The most known scenario for portability refers to the migration between Clouds (or change of the provider, e.g. in classification of use cases from [1]) of all of a part of existing services and applications. The migration's reasons vary from optimal choice regarding utilization, expenses or earnings, to technology changes or even legal issues. Due to the portability problems, rewriting the services and applications is usually required nowadays to comply with this scenario.

The migration between Clouds is just one of the scenarios in which multiple Clouds are involved. Multiple Clouds can be used, according NIST [2], simultaneously, using services from several Clouds at a time, or serially, using services from one Cloud at a time (in the case of migration between Clouds as described above, interface across multiple Clouds, or work with a selected Cloud). Most challenging in terms of portability are the scenarios in which Cloud applications are distributed across two or more providers and administrative domains simultaneously. Two different approaches are possible (Fig. 1): the portability is ensured at the Cloud provider site in the case of Federated Clouds, or outside the Cloud provider sited in the case of Hybrid Clouds.

In the case of Federated Clouds, the providers agree each other how to mutually enforce policy and governance, establishing a common trust boundary. We consider that in this case fall the Horizontal Federation (approach undertaken by Reservoir [3]), when two or more Cloud providers join to share resources

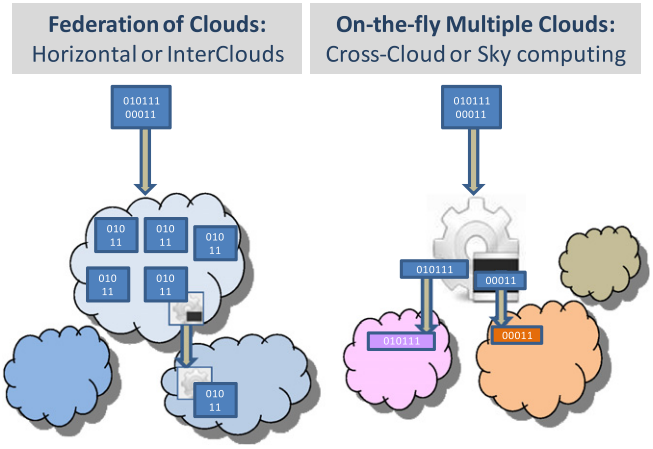


Fig. 1. Scenarios of using multiple Clouds in which portability is required.

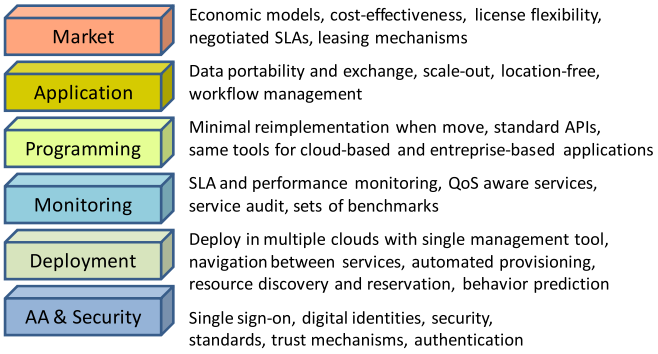


Fig. 2. Functional and non-functional requirements to allow the multiple Cloud usage.

(participants who have excess capacity can share their resources, for an agreed-upon price, with participants needing additional resources) and a logical topology is maintained regardless the physical location of the components. A second case falling in this category is that of InterCloud [4], a federation with a consensus in what concerns addressing, naming, identity, trust, presence, multicast, time domain or application messaging (without constraints on the topological information, ensuring a higher degree of dynamicity).

In the case of Hybrid Clouds (or On-demand or On-the-fly grouping), the applications are spanning over several Clouds and both administrative domains and trust boundaries are crossed. A concrete scenario is that of different Cloud services used at the same time to run tests or to build test environment. We consider that in this case fall the Cross-Cloud [5], an ad-hoc federation establishment between a Cloud needing external resources and a Cloud offering resources. A second case falling in this category is that of Sky computing [6] when resources and services from different Clouds are used to provide new services other than the ones of each individual Clouds and transparency of multiple Clouds is provided offering a single-Cloud like image (Sky providers are consumers of Cloud providers' services, offering dynamicity). Brokerage systems are usually supporting both cases.

The main requirements to support multiple Cloud usage scenarios are suggested in Fig. 2 and discussed in more details in [7].

Note that mOSAIC offer that is exposed in this paper intends to supports the on-demand grouping of the multiple Clouds services as it allows to postpone at run-time the decision of the resource selection, as well as providing a broker for selecting these resources. The requirements to be fulfilled by mOSAIC

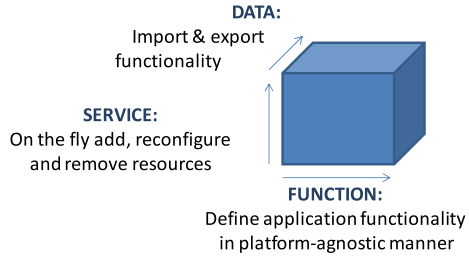


Fig. 3. Dimensions of the Cloud portability.

solutions are referring to the middle levels from Fig. 2: application, programming, monitoring and deployment. More precisely it ensures:

- scale-out at the level of application components;
- data portability using drivers for certain resource types;
- no code changes when moving application between Clouds;
- same tools for the desktop simulator and the real Cloud;
- one management tool for deployment in different Clouds;
- service level agreement (SLA) monitoring;
- automated provisioning using a Cloud agency.

2.2. Cloud portability levels

The evolution of technical solutions for the portability has encountered until now three stages [8]. First stage refers to the portability of virtual machines between Clouds, the import and export of virtual machines, the Open Virtualization Format (OVF) standard providing a good solution for this case. The second stage targeted the portability of virtual machines together with the network settings between Clouds. The current stage addresses the APIs, migration and grouping on-demand, while the portability issues have been moved to storage domain.

The types and solutions for portability can be classified in three categories according [9]: functional portability, data portability and service enhancement (see Fig. 3). The first type is achieved by defining the application's functionality details in a vendor-agnostic manner. The second type is achieved when a customer is able to retrieve application data from one provider and to import this into an equivalent application hosted by another provider. To deal with this type of portability it is needed to provide a platform-independent data representation, and to generate the specific target representations and even the code for the application's data access layer (the achievement depends on standardization of data import and export functionality between providers). For service enhancement, metadata is added through annotations, and control APIs are allowing infrastructure to be added, reconfigured, or removed in real time, either by humans or programmatically based on traffic, outages or other factors.

Note that these requirements of portability are different for each delivery model. According [10], at Software as a Service (SaaS) level, the customer is substituting an application with a new one, and portability means to preserve the functionality of the initial application and is evaluated based on proprietary or open source code, proprietary or open standard data formats, integration technologies and application server/operating system. At IaaS level, the application and its data migrate and continue to run on a new provider premises, and portability is evaluated based on ability to port virtual machines, storage, communications and the underlying configurations across infrastructure providers. At PaaS, the focus is on minimizing the amount of rewriting when the application is ported, while preserving or enhancing controls; the portability is evaluated based on proprietary or open source programming languages for application development, proprietary

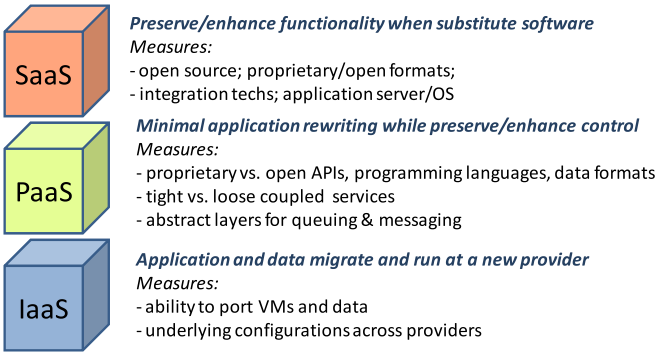


Fig. 4. Cloud portability at XaaS level.

or open data formats, tight integration or loose coupled services, abstraction levels for queuing and messaging services (see Fig. 4).

We considered our mOSAIC solution to be representative for the latest development phase of portability, as it supposes the existence of external solutions for portable virtual machines, allowing the on-demand resource provisioning, and proposing a high level abstraction of Cloud resources. Moreover, it targets the coverage of all faces of the cube from Fig. 3. More precisely it ensures:

- functional portability through the vendor-agnostic API;
- data portability through the use of drivers for the same type of data (unified resource representation);
- service enhancement through the facilities of the platform to add and remove components during the execution of an application.

mOSAIC is dealing with portability at IaaS and PaaS level and it tries to keep the characteristics of a PaaS (e.g. resource and deployment transparency), to offer an open source set of APIs for application development and abstraction layers for different types of Cloud resources (including queuing and messaging services).

2.3. Current technical solutions to ensure portability

In the case of a portable application build from components, its components should be able to be easily moved and reused regardless of the provider, location, operating system, storage, format or API. In order to achieve this goal, generalized abstractions between the application logic, data and system interfaces are needed to be defined. In this context, the portability approaches can be classified as building or using open APIs, protocols and standards, layers of abstractions and semantic repositories.

Open APIs are, for example, jclouds for Java, libcloud for Python, SimpleCloud for PHP or Dasein Cloud for Java. All of these are language dependent and offer a thin abstraction level towards the unique representation of the resources (wrappers solutions). More complex APIs are provided by OpenNebula, Appistry, or OpenStack. All of these solutions can be classified in three categories, according [11]: API with multiple independent implementations (like Eucalyptus and EC2, AppEngine and AppScale), API runnable on multiple Clouds not necessarily through multiple independent implementations (e.g. several implementations of MapReduce), API that allow to separate the application into application-logic layer from the Cloud layer. The latest, most general, option requires a time and complexity investment by a developer to initially create the layers and further maintain them over time as the APIs change.

Open protocols are, for example, DeltaCloud or OCCI, both for Http. DeltaCloud abstracts the differences between diverse Clouds for the case of computational resources. OCCI is a specification for remote management of Cloud infrastructure, allowing the development of tools for common tasks including deployment, autonomic scaling and monitoring, and its API supports three concepts: compute, storage and network. Open standards are, for example, OVF, that describes virtual appliances to be deployed across different virtualization platform, and CDMI, that specifies the functional manner on how applications should operate with data from the Clouds.

Concerning the new levels of abstraction introduced with the advent of Cloud computing it should be noticed at least the efforts of Reservoir [3]: new actors, the service providers are mediators between infrastructure providers and end-users (single clients, businesses), while service manifestos, defining contracts and SLAs, play a key role in the architecture.

The semantics can be applied to the interface, component or data level using a unified Cloud resource model. Moreover, the semantics can be used to annotate services and applications (more precisely the appropriate properties in order for efficient deployment and execution in the infrastructure). An implementation example of the concept is UCI (Unified Cloud Interface).

Note that comprehensive analysis of the current solutions for application portability are available also in [7,12–14].

The basic ideas of mOSAIC design that should differentiate it from this context are as follows:

- an API following in its development the hardest part, the one allowing to separate the application into application-logic layer from the Cloud layer;
- application independence both from provider but also from language and programming style;
- a Cloud ontology and a semantic resource discovery mechanism.

3. Architecture of Cloud applications

The most successful applications in Cloud computing service market are the web applications. This section analysis the architecture of the Cloud applications pointing towards the differences from the traditional web applications, in order to understand the needs in the transition process towards Clouds.

Traditional web 2.0 applications rely on a three-tiered architecture as the one in Fig. 5. This traditional architecture allows separation of presentation from logic, and of logic from data. However, it is suitable mostly for applications with a predictable number of users, following a small number of usage patterns and a reduced number of load spikes. The three-tiered architecture runs into problems with the need for high scalability and elasticity of modern web applications. Furthermore, traditional web applications use relational databases for their data tier. This database system are difficult to scale or to replace in case of failure and any change in the database schema requires some downtime. Also, performing queries on these databases is slow.

On another hand, a Cloud application is expected to have a considerable user base and unpredictable load, characterized by inconsistent usage patterns and managing large amounts of data.

To transform a web application into a beneficiary of Cloud elasticity a more advanced architectural style is needed to be adopted. We propose the architecture from Fig. 6. We have identified this multi-tiered architecture after examining some real-life, very popular, web applications that could use or are currently using Cloud computing. These included Showyou, Union Station, Reddit, LinkedIn and Facebook.

The *Presentation Tier* is common to both traditional and Cloud web applications and specify how the application interacts with the user. The presentation could be user device-specific, such as that needed for a web browser. The separation of user interface details in a separate tier allows the application to have multiple interfaces.

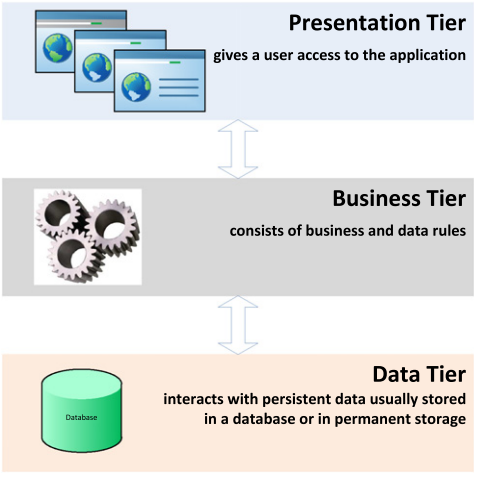


Fig. 5. Three-tiered architecture.

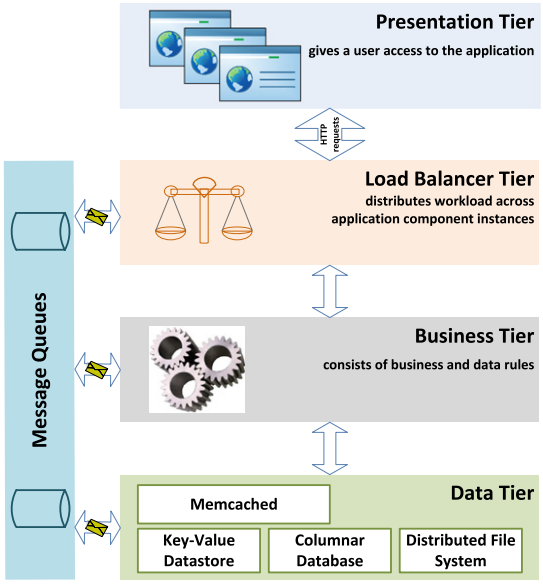


Fig. 6. Cloud multi-tiered architecture.

The *Message Queues Tier* allows to decouple the application's components. The decoupling of application into components can ensures to a certain level its scalability. A precondition for the scalability is the requirement that these components do not have tight dependences. This way, if one component fails or becomes too slow, the other components continue to work. Message queues ensure asynchronous communication between the tiers of the application and have an important role in ensuring the scalability at the level of application components. They also allow the design of multiple Cloud models in which few components run in one Cloud, possible a private one, while others use the compute power of a Public Cloud.

Once the application is scalable and every component can be scaled to multiple machines there should be also an application tier ensuring that all component instances are evenly used. The *Load Balancing Tier* distributes user requests to different component instances in order to avoid overload and minimize response time.

The *Business Tier* is responsible for implementing the business processes specific to the application use cases. The components in this tier must also be designed with scalability and robustness in mind and thus a few simple guidelines should be followed during their design. For example, the application components should share as little state information as possible because on the contrary it will be almost impossible to scale them. Another principle that should be followed is that components should be redundant such that if one component instance should fail, its workload would be immediately taken by another instance.

Just as in the traditional three-tiered architecture, the *Data Tier* in a Cloud application interacts with persistent data stored in a database or in permanent storage. But, unlike traditional architecture which use relational database systems for storing the data, in Cloud applications data is stored in highly fault-tolerant and easily scalable systems, like key-value stores, columnar databases and distributed file systems. Using such systems means also that the application will have to manage data using models quite different than those used by the relational database and will also have to account for weaker consistency models, e.g. eventual consistency. Instead, the application will benefit of excellent recoverability, durability and scalability. In order to alleviate the load on the different databases, the application could make use of memcached, an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, or other operations.

In order to ease the task of the developer of such web applications, we considered useful to provide a support for implementing the multi-tiered architecture described above. Therefore the mOSAIC API facilitates the development of portable Cloud web applications based on the multi-tiered architecture. More precisely, the API defines ways of accessing different key-value stores or columnar databases, of communicating using message queues and of building scalable, stateless, asynchronous business application components.

4. mOSAIC's application programming interfaces

In this section we present how Cloud portability was ensured by mOSAIC design and implementation and how the promises made in the previous sections are covered. While the mOSAIC APIs are the key elements in ensuring application portability, the platform as a whole provides even more features allowing application portability.

4.1. Short overview of mOSAIC as a full

mOSAIC intention is to offer answers to two different questions of the Cloud application developer concerned to ensure its application independence from a Cloud service:

1. how to port an application between Clouds;
2. which are the proper resources to run an application.

Beyond the API that helps to answer to the first question and which respects the requirements described in the previous sections, mOSAIC proposes also a software platform, a set of application tools, a semantic engine, a Cloud agency and a service discoverer. Fig. 7 enumerates the components of mOSAIC architecture. The components responsible with the Cloud portability are those grouped under application and software platform support. The other components are responsible to ensure the answer to the second question. Their description is beyond the scope of this paper.

The main design requirement is focusing on the developer of a Cloud application, who, using mOSAIC, does not need to worry about issues specific to a certain Cloud provider, but instead can focus only on its application and later deploy it on any Cloud provider he or she wishes. The main requirements of the API design are therefore the following:

- ensure the application portability;
- ensure the elasticity at application component level;
- allow the freedom to choose the programming language;
- allow to build own stack of software for an application.

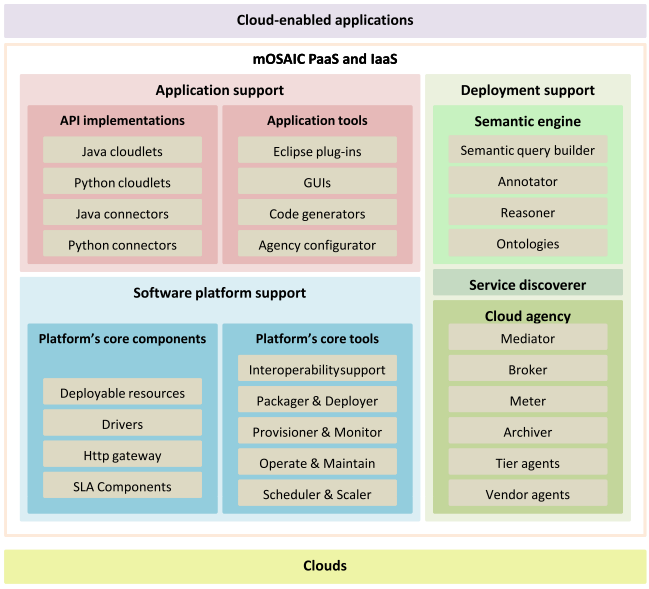


Fig. 7. Developer view: components of mOSAIC's architecture.

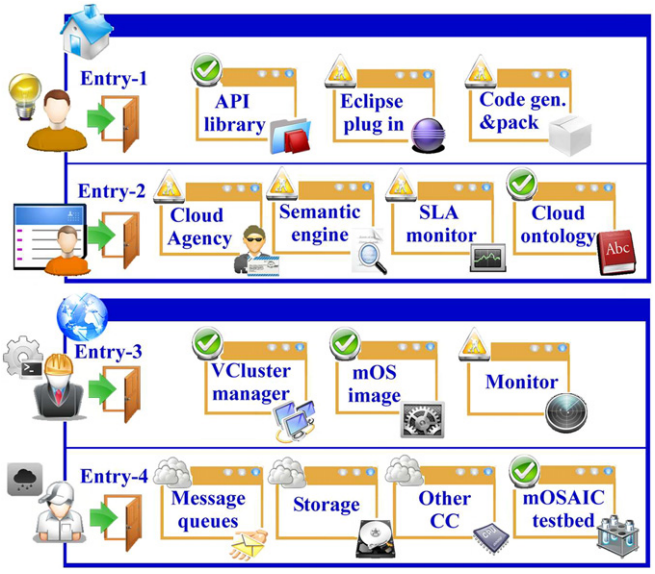


Fig. 8. User view: mOSAIC's abstraction layers and assets (their development status is marked with a 'check' or 'under construction' signs).

4.2. Levels of abstraction

mOSAIC design intends to offer a high degree of flexibility and control to the developer by introducing several levels of abstraction. On the highest level the developer lets mOSAIC control completely its application. However, it is also possible for the developer to build its application at a lower level.

As shown in Fig. 8, there are four levels of abstraction in mOSAIC, the top ones relying on the below ones. This model is compliant with the Cloud Enablement Model presented in [15] where the four tiers are Cloud Business Tier addressed by business and IT consumers, Cloud Platform Tier addressed by application developers, Cloud OS Tier addressed by IT applications and infrastructure architects, and Cloud Virtualization Tier that is addressed by data center technicians. Note that Amazon services are respecting this model too (at business tier MapReduce or billing services are available; at platform, CloudFront; at OS tier, control panels or workflows; at Virtualization tier S3, SimpleDB or EC2).

In mOSAIC, the two upper layers (Application Tools and Provisioning System) are deployed at the developer's site, while the two bottom layers (Deployment System and Resources System) are expected to be deployed on remote resources after the development phase. Depending on its intentions, the mOSAIC user can operate on each of these layers.

Application tools layer. This layer is designed for developing new applications, built from scratch. At this level, the user can choose one of the language dependent APIs (Java, Python) and implement its application. The application is built from several components with properties and relationships between them specified in an application descriptor which is used at lower levels for deploying the application and for ensuring scalability and fault-tolerance. Since an Eclipse plug-in will define graphical means for specifying parts of an application, the code generator is responsible for producing the application in a deployable form. With this layer, mOSAIC overcomes other PaaS offers by ensuring independence from the Cloud vendor, by pushing the choice for consumed Cloud services at lower layers and by delegating the resource provisioning issue at the second layer.

Provisioning system layer. This layer is designed for the owners of existing applications intending to run their applications on Cloud environments. No matter if these applications are newly developed using the tools in the upper layer or are legacy applications, they all have a common point, they all require a large number of computing or storage nodes. For legacy applications, the user has to prepare the application to support the communications according mOSAIC's rules and to produce manually the application descriptor.

The point of entrance is the Cloud agency responsible for provisioning the resources necessary for the application. The input for the agency is the application descriptor and the output is the service level agreement (SLA) and the contact lists of the resources that were booked. The agency is a complex component that ensures the brokerage of the resources and the negotiations with provider sides. Agents are dealing either with the application requirements or the provider offers [16]. In order to have a common understanding, a Cloud Ontology is used [17]. The Semantic Engine assists the discovery process on both top levels. The SLA monitor component is in contact with the remote Monitors deployed on each site (the lower level) and in case of break of the established SLA informs the Agency to take actions. Re-provisioning is possible also depending on the peaks of the applications. Like in the case of the previous layer, the decision of the Cloud services to be consumed is delegated to a lower level. Furthermore, the user is offered some control in what concerns the SLA compliance, and also the semantic assistance in service discovery is offered.

Deployment system layer. This layer is mainly designed for debugging purposes and for manual control of the resources. It is a typical entry for IaaS management and monitoring of the running applications. At this point, a Virtual Cluster Manager (VCM) is provided in order to manage all resources allocated to an application (communications, storage or processes). The VCM is active as soon as the mOSAIC Operating System (mOS) is deployed on the remote site. mOS is a small Linux distribution adapted for mOSAIC needs; currently deployable in Xen or KVM virtualization environments, like Eucalyptus or EC2. A scheduler assists the manager in deploying the processes on the booked resources. The evolution of the application and the SLA compliance is supervised by the Monitor. This will notify the VCM (for fault tolerance), the Agency (for peaks) and the home SLA Monitor (for SLA breaks) about any problems in the system. The benefits for the mOSAIC user at this level compared with other IaaS are the following:

control the processes and resources using a friendly human web interface, and automatically assistance in case of peaks and faults.

Resources system layer. This layer is designed only for resource providers for debugging and billing purposes. At this level, the native interface of each resource (not provided by mOSAIC) is used. For testing purposes, mOSAIC offers a testbed consisting in infrastructure resources of the project partners and a catalog of deployable open-source software for Cloud computing.

4.3. The concepts of the mOSAIC's API

We review here the basic concepts of the mOSAIC APIs. These concepts were first exposed in [18]. Here we provide more details.

mOSAIC-compliant applications are expressed in terms of *Cloud Building Blocks* able to communicate each other. Such a block is an identifiable entity inside the Cloud environment, and it can be either a Cloud Resource under Cloud provider control (CR) or a Cloud Component (CC).

A *Cloud Component* is a configurable building block, controlled by the application developer. It exhibits a well defined behavior, implementing functionalities and exposing them to other components. Its instances run in a Cloud environment consuming Cloud resources.

The simplest example of a CC is a Java application runnable as a service environment. It can be also more complex, like a virtual machine, configured with its own operating system, web server, application server, and with a customized e-commerce application on it. An instance of a Cloud Component in a Cloud environment resembles to an instance of an object in an object oriented application. The Components are expected to be developed following any programming language or paradigm.

The communication between Cloud Components takes places through Cloud Resources (like message queues) or through non-Cloud resources (like socket-based applications), and can use any paradigm. A Cloud Component communicates directly with a Cloud Resource, and only indirectly with another Cloud Component through the mediation of Cloud Resources, like message queues. This restriction is imposed to allow the development of services and applications using simultaneous services from multiple Cloud providers or to control the redirection of the messages in case of faults or scale-up and downs.

Most of the current web or scientific applications can be modeled in terms of components and their communications by respecting the above described constraints. However, in order to profit from the elasticity characteristic of the Cloud, their functionality should be re-considered. Take for example the e-commerce application initially deployed in one single virtual machine: deploying it on another virtual machine is easy, but the application must be rewritten for the case of more than one virtual machine or to handle a crash of a component instance.

The mOSAIC's aims to solve smoothly the problem of co-existence of multiple instances of Cloud Components of which number can vary during the application life-cycle. Its Cloud Components are *elastic, fault tolerant, manageable and autonomous*. Elasticity support is understood in terms of dealing with multiple instances of the same Component and the scale up and down of the number of instances. Fault tolerance of component's instances is achieved in an automated way (through Containers, see the concept in what follows). A Component is manageable in the sense that it is possible to configure it and to change its working parameters. The CC instances are autonomous when they are running in a Cloud environment, being independent from other Components.

Note the high level description of a Cloud application requested by mOSAIC, in terms of inter-communicating Components. This description is not affected by the way in which Components are developed (the programming language or paradigm adopted) or communicate between them (like queues or sockets).

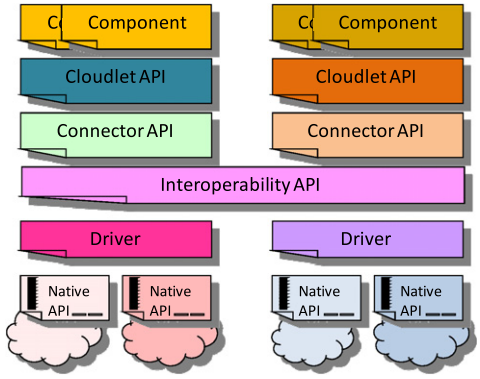


Fig. 9. Layers of the mOSAIC's of APIs supporting the Cloud components development and communications.

4.4. Unified resource representation and its achievement through mOSAIC's API layers

One of the most important concept supported by mOSAIC is that of *unified resource representation*.

In order to explain the view of mOSAIC on this concept, we provide a simple example: there are several implementation of the columnar database concept for Cloud computing (hosted solutions like SimpleDB, BigTable, or deployable solutions like Cassandra, HBase, Hypertable, Tiramola, MonetDB, LucidDB, Akiban, to name a few). From the point of view of the application the resource of columnar database should be operated in the same manner despite the differences between the implementations.

Most of the current solutions for unified resource representation can be classified as wrappers ensuring a thin layer of abstraction that remains dependent on the style of programming when interacting with the resource. mOSAIC intends to solve this problem introducing a layered set of APIs, each layer ensuring a new step in acquiring independence from the back end resource (in the sense of the language or programming style used in its API).

The APIs layers are grouped as follows (Fig. 9):

(1) *Language-dependent layers (high level).*

Cloudlet API

It is designed for creating Cloud components, in a similar way as the Java Servlet technology provides standard programming components in J2EE environments (style adopted also by Google's AppEngine). Further details are provided in the next subsection.

Connector API

It provides abstractions for the Cloud resources. The developer is allowed to touch it, depending on the programming language. It can be roughly compared with Java's JDBC or JDO. It ensures the uniformity for the programming paradigms, as all the implementations of the connector API in object oriented programming languages should have similar class hierarchies, method signatures, or patterns.

(2) *Cloudware layer (middle level).*

Interoperability API

It aims to provide programming language interoperability, protocol syntax and semantic enforcements. Is a RPC solution that abstracts addressing. It offers stubs to the Drivers, and proxies to the Connector.

(3) *Resource layers (low level).*

Driver API (wrapper)

Wraps the native API, providing the first level of uniformity: all resources of the same type are exported with the same interface (potentially with a lost of some custom resource features). To this level exchanging, for example, an HBase versus a Riak key-value store is just a matter of configuration.

Native resource API (protocol)

It is provided as a library (for Web service, RPC) by the Cloud vendors for a certain programming language, with a high degree of diversity (e.g. nothing common between CDMI and Cassandra Thrift APIs).

When the communication between two or more Cloud Components occurs, several APIs are used.

Programming model. The internal architecture of the mOSAIC API is built according to an asynchronous, event-based programming model. mOSAIC users need to comply with this model when writing their Cloudlets. While it may be more difficult to develop applications using an event-based approach, its advantages overcome its difficulties:

- Events prevent potential bugs caused by CPU concurrency of threads.
- Performance of event-based programs running under heavy load is more stable than that of threaded programs.
- Event-based programs use a single thread and thus, the programmer does not need to handle data races.

Cloudlets and connectors. The developers need to focus on Cloudlets and Connectors in order to develop their mOSAIC-compliant applications.

A Cloud Component is represented by a Cloudlet (with one or more instances) and its Container. A Cloudlet runs in a Container. A Cloudlet can have multiple instances. A Cloudlet Container hosts a single Cloudlet, even if it may host multiple instances of the same Cloudlet. The same Cloudlet may be hosted in different Containers. At runtime Cloudlet instances are non-distinguishable. When a message is directed to a Cloudlet it can be processed by any of the Cloudlet's instances. The number of instances is under control of the Container in order to grant elasticity at the Component level (with respect to the Cloudlet workload).

Each Container has a unique identifier enabling its identification at runtime. If two containers host the same Cloudlet it is possible to distinguish between them, but not between the Cloudlet instances they hosts. The number of Containers hosting the same Cloudlet can be changed in order to grant fault tolerance and eventually elasticity (programmable in this case, compared with the case of the inside of Container where is automated). The functional behavior of a Cloudlet (i.e. I/O behavior) never depends on the number of Cloudlet instances and/or on the number of Cloudlet Containers which host the Cloudlet.

The Cloudlet instances are stateless (in terms of data), even if each Cloudlet owns a Context as local data. The Context is explicitly defined by the Cloudlet developer and is local to the Cloudlet instance. It cannot and should not affect the Cloudlet functional behavior. The Context class can be used, as an example, for caching purposes or to maintain usage statistics.

A event-driven approach was adopted, and therefore the Cloudlet behavior is defined as reaction to events. The basic set of Cloudlet events are: Create, Initialize, InitializeSucceeded, InitializeFailed, Destroy, DestroySucceeded, DestroyFailed.

In order to access Cloud resources, the Cloudlets are using Connectors. A Connector (interface) define a new set of events to which a Cloudlet reacts. The Connectors abstract the access to Cloud resources (of any kind) in terms of actions and of a set of events as consequences of Connector actions or of Cloud resource operations. A Cloudlet can use even more than one Connector at the same time.

For a given Cloud resource we have a set of Connectors, each one exposing a given usage pattern (set of actions and events). A Queue Connector, for example, defines two usage patterns: consumer and publisher. The consumer role offers, to a Cloudlet which adopts it, a set of events (list) and no actions. The publisher role offers the events list and the publish action.

Since the Connectors should be able to access the Cloud resources in a transparent way, i.e. they should not be aware of the position of the Drivers and of the resources accessing them, the behavior of the Connector is considered to be of RPC type. Moreover the applications may need to interactively act and access the resources.

Interoperability API. The Interoperability API enables Connectors to invoke Driver operations, having no knowledge of the technology they support and which can resides on different machines.

While a prototypical interoperability protocol between Connectors and Drivers can be build on top of well known general-purpose interoperability protocols proposed in the last decade (like SOAP), they are excluded due to the big overhead introduced by the often Connector–Driver communications (especially in the internal communication between Cloud components). In fact the only way to avoid introducing overhead is to have a protocol which is not of general-purpose but dedicated to the given pair Connector/Driver.

The Connector abstraction is developed only once for a type of resource and should not be in principle influenced by the appearance of new Drivers. Moreover, the Drivers should be implemented in a programming language which well fit the target technology, while the Connectors are offered in a language which is related to the application developer needs. Therefore the Interoperability API offers a language decoupling between Connectors and Drivers.

Note that the Connectors and Drivers may be involved in big amount of data exchanges and, moreover, they are connected through networks which may not be fully reliable. Therefore an asynchronous protocol was considered.

More details can be found in [19].

4.5. Status of the implementation and reporting

A first implementation of the API and platform was released in the public domain in the Summer of 2011. It is available as open-source solution at https://bitbucket.org/mosaic/mosaic-java-platform. This distribution includes a short documentation and simple examples like producer–consumer exchanges, as well as one complex example of a Twitter watcher.

Java was selected as first programming language to proof the viability of the proposed conceptual model. In this context, the developer of a Cloud application can easily use Eclipse (by importing the classes available to the above mentioned address).

Moreover, to help the smooth development of the applications on developer's desktop, a specific tool is provided: mOSAIC Portable Testbed Cluster is a virtual cluster software build using VirtualBox virtualization technology. It allows the development on a desktop of the components as well as their deployment in several virtual machines running on the developer's computer. Porting the components to a Public or Private Cloud does not require any modification. It is available for download at: http://developers.mosaic-cloud.eu/confluence/display/MOSAIC/mOSAIC+Portable+Testbed+Cluster.

For the deployment of mOSAIC application on Cloud provider infrastructure, mOS is used. mOS, mOSAIC OS (mentioned earlier

in this paper), is a minimal OS created special for deployment. It is based on SliTaz Linux 3.0 and is available at http://developers.mosaic-cloud.eu/confluence/display/MOSAIC/mOS.

A demo of the platform functionality in terms of the life-cycle of the components can be found at http://web.info.uvt.ro/~petcu/demo_mosaic_19102011.avi (with reference to the Twitter watcher).

The concepts behind different sub-systems of the mOSAIC solutions enumerated in Fig. 7 were reported earlier in [18–20] as well as in the public reports available on the project site http://www.mosaic-cloud.eu. This paper presents for the first time these concepts in the context of portability problem and reveals the first experimental results.

5. Experimental considerations

5.1. Application scenario

In order to show the benefits of mOSAIC for applications following the architectural style described in Section 3, a real application scenario simulator was built. To test and prove the proposed mOSAIC API (and everything that supports the API, like framework, infrastructure middleware etc.) we implemented the simulator following first the classic three-tiered architecture described in the aforementioned section and then we implemented it using the Java version of the mOSAIC API. The classic three-tiered model was implemented following a synchronous programming model while the implementation based on mOSAIC follows an asynchronous approach based on events.

The model of the chosen application scenario is applied by the e-commerce platform that has support for online payment, namely the checkout process. Legacy implementation of the checkout process uses, in most of the cases, web services as a communication layer and synchronous model as programming paradigm. The checkout application must accomplish the following operations:

- retrieve user payment details and product list;
- calculate the total amount that needs to be charged from the user's credit card;
- charge the credit card by contacting the credit card issuer (the bank);
- save the transaction details for later use.

In the simulator each operation is assigned to a component, called from now on agent, specialized only for a particular type of operation. The underlying communication channel is based on a message queue technology implementation instead of the legacy web service technology while the details about products and transactions are stored in a key-value store instead of a relational database. Fig. 10 describes the architecture of the testing application.

In the synchronous implementation the agents work based on the following scenario:

- the *BillingAgent* receives a checkout request and now, sequentially, checks for the total price of the products in the shopping list by sending a message to the *ProductsAgent* and waits for the reply;
- the *ProductsAgent* retrieves from a key-value store the prices of the products on the shopping list, computes the total prices and sends it to the *BillingAgent* which then issues a credit card charge to the *ChargeAgent*;
- after the *BillingAgent* receives the acknowledgment of the payment from the *ChargeAgent* it stores the transaction details in a key-value store and marks the transaction as being finished.

In the synchronous implementation, the behavior of the web services is simulated but instead of having HTTP for transportation and XML for data serialization we use messages for transport and JSON encoding for data serialization. In this version each *BillingAgent* will be blocked with one transaction until the entire flow it is completed. The other depending agents, *ProductsAgent* and *ChargeAgent*, will follow the same behavior. Following a checkout flow if we have *N BillingAgents* in order to optimize the processing of the transactions the number of the *ProductsAgent* and *ChargeAgent* must be $N/2$. For accessing resources like message queues and key-value stores, this implementation uses client libraries specific to the systems used for providing the resources (here RabbitMQ, respectively Riak) and thus, this legacy approach is resource dependent and its' scalability is limited because of the direct link between the number of resources involved in the processing flow.

The scenario for asynchronous implementation is slightly different but more efficient since a single *BillingAgent* may handle several requests at the same time:

- the *BillingAgent* receives a checkout request and sends a request to the *ProductsAgent* to compute the total price; immediately after the request is sent to the *ProductsAgent*, the *BillingAgent* can start processing a new checkout request until the reply from the *ProductsAgent* arrives;
- while the *ProductsAgent* waits for product prices from the key-value store, it can also start processing other requests and when it has the total price for a transaction it just sends the result to the *BillingAgent* which then issues a credit card charge to the *ChargeAgent*;
- after the *BillingAgent* receives the acknowledgment of the payment from the *ChargeAgent* it stores the transaction details in a key-value store, but this is also done asynchronously such that if the interaction with the key-value store lasts a while, the agent is not blocked waiting for it to complete and can process other requests in the meantime.

Each agent is implemented as a Cloudlet which uses a resource type specific connector for interacting with message queues and key-value stores. In this implementation, each Cloudlet is single-threaded but a single thread can process several requests at the same time. In case of overload, we create another single-threaded Cloudlet instance and thus we can process the same number of requests as the synchronous implementation using a smaller number of threads.

5.2. Experimental results

In this section we present the results of the performed experiments, in which we compared the average throughput and latency for the two implementations, as well as the resources, in terms of number of threads, required for attaining those values. For these experiments we have used just three virtual machines (dual-core, 2 GHz CPUs, 2 GB RAM): one which runs the RabbitMQ server, one for the Riak KV store and another for running the applications.

We have run several tests on the two implementations and in each of these tests we sent about 25 000 requests to the *Billing Agents* and we measured the throughput and latency for each request.

First we have measured the throughput and latency for the synchronous implementation of the checkout process. We started experimenting with a low number of agents: 6 *Billing Agents* (BAs), 3 *Products Agents* (CAs) and 3 *Charge Agents* (CAs), each running in a separate thread. As it can be seen in Figs. 11 and 12 the throughput for this experiment is very low and the latency is very high (see results for T:6BA–3PA–3CA). Thus, we increased the number of agents and we observed that once we reach 40 BAs no improvements on the throughput can be achieved.

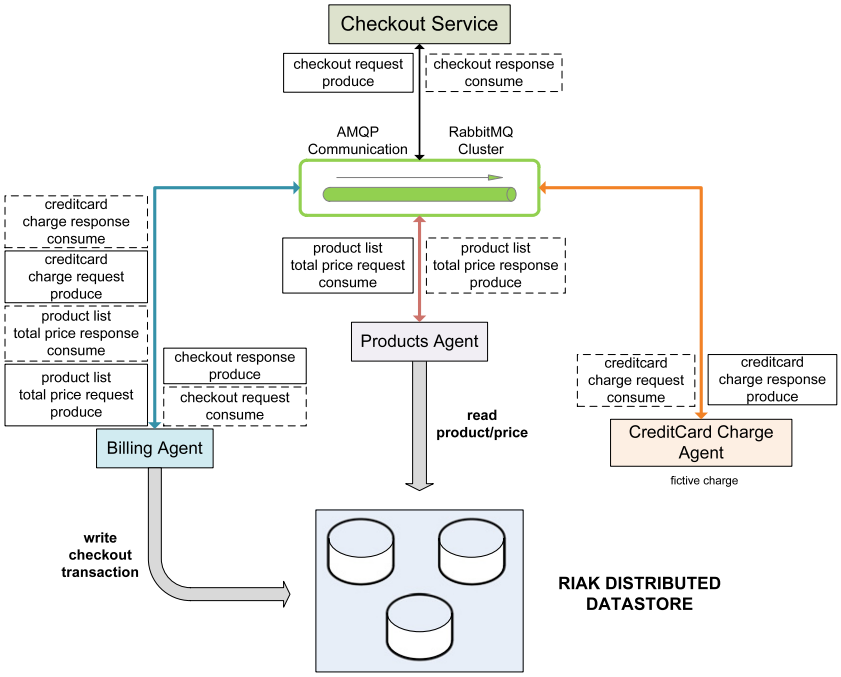


Fig. 10. Architecture of the testing application.

Table 1
Deployable open-source technologies to manage Cloud resources.

Type	Representatives
Storage resources	
Columnar databases	*Clustered*: Cassandra, Hadoop HBase, Hypertable
Key-value databases	*Clustered*: Riak KV, Membase, kumofs, Voldemort, Hibari, Oracle NoSQL *Single server*: Redis, MemcacheD *Embedded*: Tokyo Cabinet, Kyoto Cabinet, BerkeleyDB, LevelDB
Document databases	MongoDB, CouchDB
Distributed file-systems	*Clustered*: Hadoop HDFS, GridFS, Ceph, XtreemFS
Atomic databases	*Clustered*: Zookeeper, Scalaris, Keyspace
Compute resources	
Map-Reduce	*Clustered*: Hadoop, Riak Pipe, Hive, Disco
Infrastructure	*Clustered*: Eucalyptus, OpenStack, Nimbus, Tashi, OpenQRM, OpenNebula
Communication resources	
Message queues	*Clustered:* RabbitMQ, Kestrel, HornetQ *Single server:* ActiveMQ *Embedded*: ZeroMQ
RPC queues	*Single*: Gearman, beanstalkd
General tools	
Monitoring	collectd, Ganglia, Hyperic, Nagios
Logging	*Clustered*: Scribe, Logplex, Flume

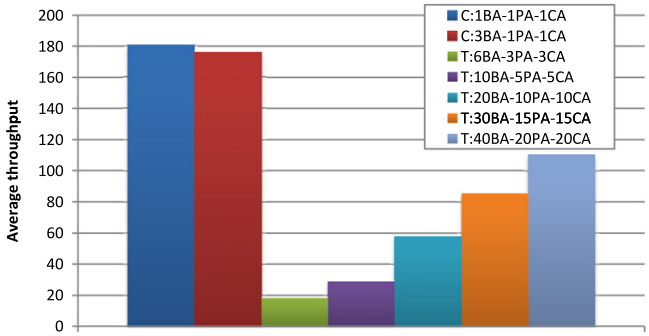


Fig. 11. Asynchronous vs. synchronous implementation: average throughput.

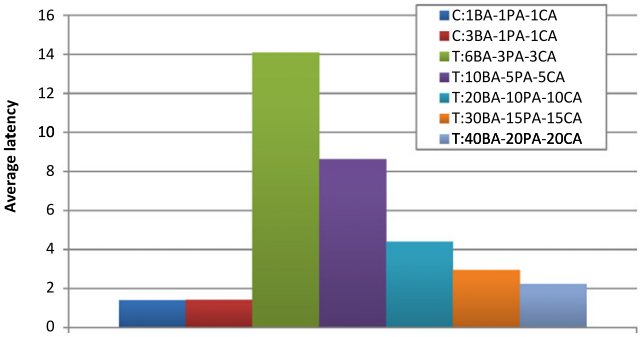


Fig. 12. Asynchronous vs. synchronous implementation: average latency.

Next, we have measured the throughput and latency for the asynchronous version. Since we have used only a limited number of virtual machines and only one of them was running the actual application, we have chosen to run experiments only for the cases when we have 1 Cloudlet for each of the *Billing Agent*, *Product Agent* and *Charge Agent* (experiment C:1BA–1PA–1CA in the figures) and for when we have 3 Cloudlets for the *Billing Agent* and 1 Cloudlet for each of the other two (experiment C:3BA–1PA–1CA in the figures). For this application this seems to be the limit of Cloudlet instances running on the same machine, as even if we increased the number of instances we did not observed any improvement of the two parameters. However, if several other virtual machines would be available, we could benefit of the inherent scalability of the asynchronous version. But even with this limited number of Cloudlet instances, which also means a limited number of threads (each Cloudlet instance is single-threaded), one can easily observe from Fig. 11 that the throughput is almost double than the best result obtained in the synchronous experiments. These is due to the fact that the *Billing Agent* Cloudlet can handle more than 40 requests in parallel, which is the limit for the synchronous version. Furthermore, Fig. 12 shows that the latency is also better in the asynchronous implementation (almost 40% better). This can be explained by the way the *Products Agent* works in the two implementations. In the synchronous version, when the agent receives a request to compute the total price of a transaction it sequentially asks the KV for the price of each product in the request and waits for each price. The asynchronous *Products Agent* can process requests for several transactions in parallel and for each transaction the prices are retrieved for the KV store in parallel.

6. Related Cloud technologies

6.1. Re-use of deployable systems

mOSAIC intends to provide a deployable system as open-source solution. In the current Cloud market there is already a tremendous number of deployable technologies for Cloud computing. In Table 1 we consider the open-source ones (we consider useful to present these since a such classification is not yet available). mOSAIC intends to build wrappers (in form of Drivers) for different open-source deployable management systems of Cloud resources (that are exposed in the above mentioned table). Currently, drivers are available for Riak, Membase, Redis and RabbitMQ. Eucalyptus is used as infrastructure resource manager for testing purposes, being compatible with Amazon technologies.

While different technologies are focusing in specific areas, mOSAIC intends to relay on their availability and to offer an upper layer that encompass them, in order to provide a comprehensive offer for the application developer (covering several types of resources).

Connectors and drivers to few hosted systems are expected to be developed. For example, Amazon S3 is one system for which a Driver was build. In analogy with the previous list, Table 2 indicates most known systems.

6.2. Positioning versus other APIs which provide an unified view of the Cloud resources

As mentioned already the Driver API of mOSAIC is a concept similar to some of the existing libraries or frameworks. We can mention here DeltaCloud, DataNucleus, jclouds, JetS3t, Spring Data, libcloud, CloudLoop, Dasein Cloud API, Sun Cloud API, Simple Cloud API, typica, Restlet, BOOM, SalsaHpc, Dryad, or Orleans. Most of them are acting as wrappers of other technologies (e.g. this is the case for example for libcloud, DeltaCloud, Cloudloop, DataNucleus, Dasein, jclouds, or SimpleClouds). They propose a uniform API to different services, but maintain in most cases the programming style (e.g. Http requests). This is also the case of the implementations of the emerging standardized APIs, OCCI for virtual machine management, and CDMI for data management. From the point of view of mOSAIC these wrappers can be seen as existing Drivers to which Connectors should be build. The mOSAIC's Connectors offers the chance to eliminate the dependence on the programming style of the underlying Native or Driver APIs.

6.3. Positioning versus other PaaSs

The number of Platform-as-a-Service offers is small compared with IaaS or SaaS ones. Table 3 enumerates part of them.

The closest PaaS from the point of view of enabler of mixing and wrapping Cloud aware technologies are DotCloud and Stackato. DotCloud lets create, as mOSAIC does, the software stack for a certain applications, but is based only on wrappers and the multiple Cloud and SLA support are not open source. Stackato, an extension of CloudFoundry based on VMware technologies, is a hosted systems that targets only web applications.

Most of the PaaS do not consider the portability issue between different IaaS services. This is the case of the following. AppEngine, SmartPlatform, Heroku and Duostack are hosted systems that are targeting web applications build on a restricted stack of software. The same is valid also for Azure and XAP. AppScale and TyphoonAE are extensions of Google's AppEngine targeting also web applications and Python. Cast is an API-driven system and the functionality is exposed through a JSON REST API. NodeJitsu has an RESTful API and communicates via JSON, the application type being compliant with the concepts of Node.js (scalable applications in JavaScript). CloudBees is designed for Java applications, while OpenShift is designed for Perl, PHP, Python, Ruby and Java (latest not free) applications.

Few PaaS offer are dealing with multiple Cloud. SpawnGrid is sustaining horizontal federations. Several European projects are currently working to offer PaaS for federations of Clouds [21]. We have mention in Table 3 some of them: Contrail, OPTIMIS, 4CaaST, TClouds and BonFIRE. Note that Cloud4SOA is working to ensure the portability between PaaS using a semantic approach. mOSAIC is promising to serve the more complex case of multiple Clouds, the Sky computing.

7. Conclusions and future work

While different approaches for portability across Clouds already exist, they are not adopted on a wide scale due to their focus on specific topics, the resistance of the resource providers in keeping the advantage of being unique, or gaps in the technological offers. In particular, the open APIs have a key role in achieving portability, but most of the current offers are related to wrappers for existing provider services forcing the developer to comply with a certain programming style or language.

We proposed a layered set of APIs that are offering a supplementary degree of freedom, from programming languages and style. The specifications of these APIs and a proof-of-the-concept implementation in Java is already available on public open-source repositories, and at least one more will be available soon. Similar proposals targeting the paradigm-free programmable use of Cloud services were not registered until now. Another degree of freedom in the portability of application context is provided by the open-source deployable Cloudware allowing to postpone the selection of the Cloud resources at deployment time, by exploiting the benefits of technologies build for agent systems and semantic processing. The Cloudware provides at this moment

Table 2
Hosted systems manage Cloud resources.

Type	Representatives
Storage resources	
Columnar databases	Amazon SimpleDB, BigTable
Distributed file-systems	Amazon S3, RackSpace Cloud Files
Compute resources	
Map-Reduce	Amazon Elastic Map Reduce
Infrastructure	Amazon EC2, Amazon EBS, RackSpace Cloud Servers, GoGrid, RightScale, FlexiScale, Slicehost, ElasticHosts, CloudSigma, NewServers, Tapp, SlapOS, Elastic Server, Enomaly
Communications resources	
Message queues	Amazon SQS, Amazon SNS, StormMQ
General tools	
Monitoring	New Relic
Logging	Loggly

Table 3
PaaS offers: hosted, deployable (open source) and promised by on-going European projects.

Type	Representatives
Hosted	AppEngine, SmartPlatform, Heroku, Azure, GigaSpaces XAP, Stackato, Bungee Connect, Aneka
Deployable	AppScale, TyphoonAE, Cast, Project Caroline, SpawnGrid, Duostack, OpenShift, CloudFoundry, NodeJitsu, Nodester, CloudBees, DotCloud
Promised	CONTRAIL, OPTIMIS, 4CaaSt, TClouds, BonFIRE, Cloud4SOA, mOSAIC

only a minimal functionality; it is scheduled to be completed in the next two years.

The novelty of the API proposal consists in: (a) the concepts of Cloud Component, Cloudlet, Connector, Container, and Interoperability API; (b) the unified representation of several types of Cloud resources; (c) the new API layers that allow not only independence from the Cloud providers APIs but also from the language and programming style; (d) the implementation and integration in a open source and deployable PaaS solution.

The main characteristics of mOSAIC software platform differentiating it from other PaaSs are the following: (1) is an open-source platform based on a stack of existing Cloud technologies; (2) deals with Sky computing scenarios, as the most general case of multiple Cloud grouping; (3) targets the development of Cloud application on top of Private of Public Cloud resources; (4) is a deployable Cloudware; (5) lets create a software stack appropriate for a certain application; (6) does not impose strong constraints on the type of applications to be deployed; (7) allows to postpone the decisions concerning the resource providers until deployment phase.

The architecture of a Cloud application that fits at the best the mOSAIC solution was also discussed from conceptual as well as experimental point of view. Note also that, as being a deployable system, mOSAIC is well suited for Private Clouds.

mOSAIC solutions are in an early stage of development. A new implementation of the API and the enhancement of the platform features are ongoing tasks. Considerable effort will be invested in the near future to ensure the intensive testing, benchmarking, comparisons with other systems, stability of all sub-systems as well as the development of new Drivers, Cloudlets, Connectors, and proof-of-the-concept applications.

Acknowledgments

The development of mOSAIC is being done by a large multinational team that is not reflected fully in the list of authors of this paper. We thank the entire team and especially Massimilliano Rak for API concepts, Iñigo Lazcanotegui Larrarte for leading the API design, and the group of Institute e-Austria Timişoara for implementing the concepts.

Appendix. References for the technologies

The following tables presents the references to the technologies referred in this paper.

Acronym	Http reference
4CaaSt	http://4caast.morfeo-project.org/
ActiveMQ	http://activemq.apache.org/
Akiban	http://www.akiban.org/
Amazon EBS	http://aws.amazon.com/ebs
Amazon EC2	http://aws.amazon.com/ec2
Amazon EMR	http://aws.amazon.com/elasticmapreduce
Amazon S3	http://aws.amazon.com/s3
Amazon SimpleDB	http://aws.amazon.com/simpledb
Amazon SNS	http://aws.amazon.com/sns
Amazon SQS	http://aws.amazon.com/sqs
Apache Hadoop	http://hadoop.apache.org/
Apache Libcloud	http://libcloud.apache.org/
Aneka	http://www.manjrasoft.com/products.html
AppEngine	http://code.google.com/appengine
AppScale	http://appscale.cs.ucsb.edu/
Appistry	http://www.appistry.com
Azure	http://www.microsoft.com/windowsazure
beanstalkd	http://kr.github.com/beanstalkd
BigTable	http://labs.google.com/papers/bigtable.html
BonFIRE	http://www.bonfire-project.eu/
BOOM	http://boom.cs.berkeley.edu/
Bungee Connect	http://www.bungeeconnect.com/
Cassandra	http://incubator.apache.org/cassandra
Cast	http://cast-project.org/
CDMI	http://www.snia.org/cdmi
Ceph	http://ceph.newdream.net/
Cloud4SOA	http://www.cloud4soa.eu/
CloudBees	http://www.cloudbees.com/

Acronym	Http reference
CloudFoundry	http://www.cloudfoundry.com/
CloudLoop	http://www.java.net/project/cloudloop
CloudSigma	http://www.cloudsigma.com/
collectd	http://collectd.org/
CONTRAIL	http://contrail-project.eu/
CouchDB	http://couchdb.apache.org/
Dasein Cloud API	http://dasein-cloud.sourceforge.net/
DataNucleus	http://www.datanucleus.org/
DeltaCloud	http://incubator.apache.org/deltacloud/
Disco	http://discoproject.org/
DMTF	http://www.dmtf.org/
DotCloud	http://www.dotcloud.com/
Dryad	http://research.microsoft.com/en-us/projects/dryad
Duostack	http://www.duostack.com/
ElasticHosts	http://www.elastichosts.com/
Elastic Server	http://elasticserver.com/
Enomaly	http://www.enomaly.com/
Eucalyptus	http://www.eucalyptus.com/
FlexiScale	http://www.flexiant.com/products/flexiscale
Ganglia	http://ganglia.info/
Gearman	http://gearman.org/
GoGrid	http://www.gogrid.com/
GridFS	http://www.mongodb.org/display/DOCS/GridFS
Hadoop	http://hadoop.apache.org/
Hadoop HBase	http://hbase.apache.org/
Hadoop HDFS	http://hadoop.apache.org/hdfs
Heroku	http://heroku.com/
Hibari	http://hibari.sourceforge.net/
Hive	http://hive.apache.org/
HornetQ	http://jboss.org/hornetq.html
Hyperic	http://www.hyperic.com/
Hypertable	http://www.hypertable.org/
jclouds	http://www.jclouds.org
JetS3t	http://jets3t.s3.amazonaws.com/toolkit/toolkit.html
Kestrel	http://github.com/robey/kestrel
Keyspace	http://scalien.com/
kumofs	http://kumofs.sourceforge.net/
Kyoto Cabinet	http://fallabs.com/kyotocabinet
LevelDB	http://code.google.com/p/leveldb
libcloud	http://incubator.apache.org/libcloud
Loggly	http://loggly.com/
Logplex	http://github.com/heroku/logplex
LucidDB	http://www.luciddb.org/
Membase	http://membase.org/
MemcacheD	http://memcached.org/
MemcacheDB	http://memcachedb.org/
MonetDB	http://monetdb.cwi.nl/
MongoDB	http://www.mongodb.org/
mOSAIC	http://www.mosaic-cloud.eu/
Nagios	http://www.nagios.org/
New Relic	http://newrelic.com/
NewServers	http://www.newservers.com/
Nimbus	http://www.nimbusproject.org/
NodeJitsu	http://www.nodejitsu.com/
Nodester	http://nodester.com/
OCCI	http://occi-wg.org/
OpenNebula	http://opennebula.org/
OpenQRM	http://www.openqrm.com/
OpenShift	http://openshift.redhat.com/
OpenStack	http://www.openstack.org/
OPTIMIS	http://www.optimis-project.eu/
Oracle NoSQL DB	http://www.oracle.com/technetwork/database/nosqldb
Orleans	http://research.microsoft.com/en-us/projects/orleans
Project Caroline	http://www.projectcaroline.net/
RabbitMQ	http://www.rabbitmq.com/
RackSpace Cloud Files	http://www.rackspacecloud.com/
Redis	http://code.google.com/p/redis
Restlet	http://www.restlet.org/
Riak KV	http://riak.basho.com/
Riak	http://wiki.basho.com/display/RIAK/
Map-Reduce	MapReduce
Riak Pipe	http://github.com/basho/riak_pipe
RightScale	http://www.rightscale.com/
SalsaHpc	http://salsahpc.indiana.edu/
Scalaris	http://code.google.com/p/scalaris
Simple Cloud API	http://simplecloud.org/
SlapOS	http://www.slapos.org/
Slicehost	http://www.slicehost.com/
SmartPlatform	http://www.joyent.com/technology/smartplatform
SpawnGrid	http://spawngrid.com/
Spring Data	http://www.springsource.org/spring-data
Stackato	http://www.activestate.com/cloud
StormMQ	http://stormmq.com/
Sun Cloud API	http://kenai.com/projects/suncloudapis
Tapp	http://www.tapp.in/
Tashi	http://incubator.apache.org/tashi
TClouds	http://www.tclouds-project.eu/
terrastore	http://code.google.com/p/terrastore
Tiramola	http://code.google.com/p/tiramola
Tokyo Cabinet	http://fallabs.com/tokyocabinet
TyphoonAE	http://code.google.com/p/typhoonae
typica	http://code.google.com/p/typica
Voldemort	http://project-voldemort.com/
XAP	http://www.gigaspaces.com/xap
XtreemFS	http://www.xtreemfs.org/
ZeroMQ	http://www.zeromq.org/
Zookeeper	http://hadoop.apache.org/zookeeper

References

[1] G. Sperb Machado, D. Hausheer, B. Stiller, Considerations on the interoperability of and between Cloud computing standards, in: Proceedings, OGF27, G2C-Net, 2009.

[2] N.W. Group, Nist Cloud computing standards roadmap-version 1.0, Special Publication 500-291, December 2011.

[3] B. Rochwerger, D. Breitgand, A. Epstein, D. Hadas, I. Loy, K. Nagin, J. Tordsson, C. Ragusa, M. Villari, S. Clayman, E. Levy, A. Maraschini, P. Massonet, H. Munoz, G. Tofetti, Reservoir—when one Cloud is not enough, Computer 44 (2011) 44–51.

[4] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, M. Morrow, Blueprint for the interCloud-protocols and formats for Cloud computing interoperability, in: Proceedings, ICIW'09, IEEE Computer Press, 2009, pp. 328–336.

[5] A. Celesti, F. Tusa, M. Villari, A. Puliafito, Three-phase cross-Cloud federation model: the Cloud sso authentication, in: Proceedings, AFIN 2010, IEEE Computer Press, 2010, pp. 94–101.

[6] K. Keahey, M. Tsugawa, A. Matsunaga, J. Fortes, Sky computing, Internet Computing 13 (2009) 43–51.

[7] D. Petcu, Portability and interoperability between Clouds: challenges and case study, in: Proceedings of ServiceWave 2011, in: LNCS, vol. 6994, Springer-Verlag, 2011, pp. 62–74.

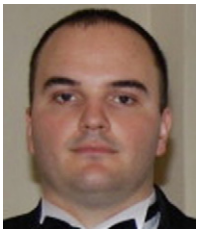
[8] J.L. Williams, An implementor's perspective on interoperable Cloud APIs, Cloud interoperability roadmaps session, OMG Technical Meeting, December 2009.

[9] K. Oberle, M. Fisher, ETSI Cloud—initial standardization requirements for Cloud services, in: Proceedings of Gecon 2010, in: LNCS, vol. 6296, 2010, pp. 105–115.

[10] C.S. Alliance, Security guidance for critical areas of focus in Cloud computing, Tech. Rep., November 2011.
[11] Z. Hill, M. Humphrey, Csal: a Cloud storage abstraction layer to enable portable Cloud applications, in: Proceedings, CloudCom 2010, IEEE Computer Society, 2010, pp. 504–511.
[12] A.L. Craig, An open Cloud computing interface status update, Cloud interoperability roadmaps session, OMG Technical Meeting, December 2009.
[13] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, K. Tarabanis, Towards a reference architecture for semantically interoperable Clouds, in: Proceedings, CloudCom 2010, IEEE Computer Society, Washington, DC, USA, 2010, pp. 143–150.
[14] N. Loutas, E. Kamateri, T. Konstantinos, Cloud computing interoperability: the state of play, in: Proceedings, CloudCom 2011, IEEE Computer Society, 2011.
[15] E. van Ommeren, S. Duivestein, J. deVadoss, C. Reijnen, E. Gunvaldson, Collaboration in the Cloud-How Cross-Boundary Collaboration is Transforming Business, Uitgeverij kleine Uil, 2009.
[16] S. Venticinque, R. Aversa, B. Di Martino, D. Petcu, Agent based Cloud provisioning and management. Design and prototypal implementation, in: Proceedings, CLOSER 2011, SciTe Press, 2011, pp. 184–191.
[17] F. Moscato, R. Aversa, B. Di Martino, T. Fortis, V. Munteanu, An analysis of mosaic ontology for Cloud resources annotation, in: Proceedings, FedCSIS 2011, IEEE Computer Press, 2011, pp. 973–980.
[18] D. Petcu, C. Craciun, M. Rak, Towards a cross platform Cloud API—components for Cloud federation, in: Proceedings, CLOSER 2011, SciTe Press, 2011, pp. 166–169.
[19] D. Petcu, C. Craciun, M. Neagul, I. Lazcanotegui, M. Rak, Building an interoperability API for sky computing, in: Proceedings, InterCloud 2011, IEEE Computer Society, 2011, pp. 405–412.
[20] D. Petcu, C. Craciun, M. Neagul, S. Panica, B. Di Martino, S. Venticinque, M. Rak, R. Aversa, Architecturing a sky computing platform, in: Proceedings of ServiceWave 2010 Workshops, in: LNCS, vol. 6569, 2011, pp. 1–13.
[21] D. Petcu, J.L. Vazquez-Poletti (Eds.), European Research Activities in Cloud Computing, Cambridge Scholars Publishing, 2012.

Dana Petcu is Director of Computer Science Department and the private research Institute e-Austria Timişoara. Her research experience is related to the topics of parallel and distributed computing, and computational mathematics. She has authored more than 250 reviewed articles and 10 textbooks. She is chief editor of the international journal Scalable Computing: Practice and Experiences and co-editor of over 15 conference proceedings. She leads three European Commission's projects in software services, HPC and Cloud computing and was involved as team leader in more than ten others related to distributed and parallel computing. She is leading scientifically the mOSAIC project and other FP7 projects (e.g. HOST, SPRERS) and several Romanian research projects. She received the international Maria-Sibylla-Merian award for women in science and an IBM award, and is nominated Romanian representative in FP7-ICTC, eIRG and eIPF.

Georgiana Macariu received her M.Sc. in Computer Science in 2008 and Ph.D. degree in 2011 at the "Politehnica" University of Timişoara, Computer Science Department, where she is currently assistant professor. She carries out also her research activities at the research Institute e-Austria Timişoara. Her area of interest covers distributed computing with a focus on creating a framework for dynamic composition of computational services tailored for large scale systems, and embedded systems with a particular interest in task scheduling on multi-core mobile communication systems.



Silviu Panica has more than eight years of experience in distributed computing being involved in several European funded projects related to grid computing and cloud computing. He was Research Assistant in the Computer Science Department of West University of Timişoara from 2005 until 2007. During this period his research interest was in remote sensing field using Grid computational power combined with the high performance infrastructure to optimize the overall processing time of the satellite images. Starting with 2007 he is a Ph.D. student at West University of Timişoara. His current research interest is in Cloud computing area focusing on the resource identification protocols applied in heterogeneous unstable distributed systems.

Ciprian Crăciun is Ph.D. student at the Faculty of Mathematics and Computer Science, West University of Timişoara and researcher at Institute e-Austria Timişoara. His current research interests are focused on distributed data management, focusing mainly on Cloud computing.