# TOWARDS A CROSS PLATFORM CLOUD API
## *Components for Cloud Federation*

Dana Petcu, Ciprian Crăciun
*Institute e-Austria Timişoara & West University of Timişoara, Romania*

Massimiliano Rak
*Second University of Naples, Italy*

Keywords: Cloud Computing, PaaS, Component Oriented Programming, API

Abstract: Cross platform APIs for cloud computing are emerging due to the need of the application developer to combine the features exposed by different cloud providers and to port the codes from one provider environment to another. Such APIs are allowing nowadays the federation of clouds to an infrastructure level, requiring a certain knowledge of programming the infrastructure. We describe a new approach for a cross platform API that encompass all cloud service levels. We expect that the implementation of this approach will offer a higher degree of portability and vendor independence for Cloud based applications.

# 1 INTRODUCTION

Cloud computing promises to enable on-demand access to a large pool of resources that can be rapidly provisioned and released with minimal management effort or service provider interaction. Supposing that this promise is kept, the adoption of the relatively new distributed computing model hardly depends on the simplicity and usefulness of the interfaces proposed to the potential users.

Cloud application programming interfaces specify how software applications interact with a cloud-based platform where these applications can be deployed. They offer ways by which the applications can request information from the platforms and use their facilities. Cloud APIs are currently exposing their interfaces using Web technologies (e.g. REST or SOAP based services) or high level programming languages.

Looking from the perspective of the current user-provider or application-cloud interaction we see two types of cloud APIs: cloud provider ones and cross platform ones. Moreover, as already happened with new emerging technologies, providers develop a large variety of proprietary solutions. These solutions face and mixe different problems (from authentication mechanisms to resource management) reflecting different interpretation of the new concept. Unfortunately their users and applications are vendor-locked until the portability problem is solved. On the other hand, the fresh introduced and few cross platform APIs attempt to abstract the details of implementations. It is expected that using a cross platform API the developer of an application calls a common and unified API and get a standard based answer regardless of the implementations of different providers. The most important benefits of this approach are the platform independence stimulating the offert of the service market, the possibility to innovate by combining different providers services, as well as lower the costs of the application implementation.

Following another dimension, that of the cloud provisioning models, we can identify three cloud APIs categories. The infrastructure ones are dealing with the provisioning and configuration of resources. The platform ones are providing a higher abstract level, offering platform capabilities in terms of services. The application ones are allowing to interface and extend application running on clouds.

We argue that, until now, cross platform APIs were produced only for infrastructure provisioning model. We mention here the recent proposals of the standardization groups, like OCCI and UCI, open-source solutions, like libcloud, jClouds, SimpleAPI or OpenNebula, and commercial ones, like DeltaCloud. They are designed either to interface only with Java, Python or PHP, or they are providing connectors or wrappers to a small number of provider offers.

We consider that in order to eliminate the vendor-

lock in problem, a new approach for a cross platform API is needed tackling with a common set of interfaces for all provisioning levels. The API should be not only platform independent but also language independent. A such approach needs to be based on a common understanding of terms, notions and relationships involved in cloud platform usage, mainly a cloud ontology. In order to proof the concepts, we intend to build both a software platform and a cloud platform, conceived as a thin layer, a middleware, which lives on the top of the cloud providers in order to enable/help the federation approach. Supplementary to the API implementation, the middleware should allow the selection of the cloud providers according to the needs of the applications, preferable at run-time, based on contractual agreements, and even to allow re-negotiations.

Following these beliefs, a development activity has recently started in the frame of mOSAIC project (www.mosaic-cloud.eu). In this paper we describe the basic concepts and requirements of the cross platform API that we propose. Next section describes the terms. Section 3 is providing more details about the components that are expected to be described in a Cloud-based applications. Section 4 is discussing the layers of the API set. Finally some conclusions and future steps are provided.

## 2 API REQUIREMENTS AND ARCHITECTURE

Before presenting the proposed API, we shall first try to clarify the meaning of API in the context of cloud computing, and what are the expectations of different parties from such an API.

From a pure software engineering point of view we have two API categories exposing operations for a certain resource: *in-process APIs* and *remote APIs*. The first one is what any developer uses on a daily basis, a combination of objects, methods, functions, or procedures (depending on the paradigm of the programming language) abstracting the resources in terms of memory usage, mutable of immutable, usually transparent, data-structures and usually opaque pieces of machine executable or interpreted code; thus using such an API is very comfortable for the developer and most of the times extremely efficient. On the other side, surpassing a single process border and bridging applications, we have the remote APIs, either in the form of Web services (like SOAP or REST based), remote calls (like Sun RPC, Java RMI, AMF), message passing (like AMQP, STOMP), or application dependent protocols (like FTP, SNMP). These APIs are abstracting the resources as reactive (request/reply) opaque agents that communicate based on fully transparent immutable data structures (like XML, JSON, ASN.1, other binary serialization). Interoperability is favored over efficiency. A drawback is the fact that remote APIs are harder to use, needing wrappers of the first kind of APIs.

If we follow a quick survey of the APIs (see e.g. related work section) exposed by the most prominent IaaS cloud providers (like Amazon or RackSpace) or proposed by the IaaS cloud standards (like OCCI or CDMI), we observe that they fit exclusively in the second category, that of remote APIs, and more specifically are of the Web services flavor, either SOAP based (like Amazon ones), or REST based (like in OCCI or CDMI). In what concerns the current state of the interoperability and the programming language independence requirements, it should be noticed that several companies or open-source projects are providing custom bindings for a wide range of programming languages. But even though at the network level (data encoding, framing, message sequence, etc.) there are standardization bodies, no standardization effort is targeting the in-process APIs. Worst, most libraries either incompletely implement the proposed standards or try to wrap the lowest denominator of the competing provider APIs. As a consequence a cloud application developer is faced with a dilemma, being forced to either choose a feature-full library created by the chosen cloud provider targeting only his own resource interfaces (getting locked in), or chose a lowest denominator, standards compliant resource API, in a rather incomplete implementation state.

On the other hand if we extend the definition of Cloud computing to include self-managed distributed data resources – like for example the Cassandra clusters from Facebook, the MongoDB shards from SourceForge, or the Hypertable deployment of Baidu – and surveying their interfacing solutions, we observe that, again exclusively, they are in the opposite corner of the remoting domain, employing RPC solutions (e.g. Thrift, Avro, Protocol Buffers, or custom binary protocols). The reason for such a decision is the need for performance and advanced features against interoperability or a common data model, and, as a consequence, a standardization process is not at all possible or even desired (at least in the near future).

Thus going back to the developer of a cloud application that might have the task not only to mix resources from different cloud providers, but also to interact with those exposed by such self-hosted distributed systems, we observe that currently no single library or standard covers such a situation.

Our platform is targeted mainly at cloud applica-

tion developers, so we intend to offer them mainly a unified in-process cloud API, but also a remote one through so-called interoperability API. The first one should be based on the emerging cloud remote API standards we have mentioned previously (OCCI, CDMI), but should go also one step further providing a feature-full unified API.

The unification aspect is threefold: (a) once we try to unify both provisioning APIs (i.e. OCCI) and data access APIs (i.e. CDMI) under the same library with common concepts and a common programing patterns; (b) second we intend to find a common ground between real cloud resources (like Amazon EC2 S3 SimpleDB, RackSpace WebFiles, Google BigTable, etc.) and self-managed distributed data resources (like Riak, Cassandra, Hypertable, HBASE, etc.) thus exposing the same categories of data models and access patterns; and (c) by providing libraries for multiple programming languages (mostly object oriented like Java, Python, etc.) we achieve uniformity from one development platform to another.

Two other goals that need to be met, and which are in conflict with the uniformity approach, are: (1) the minimization of the efficiency impact of the overall API calls (e.g. latency, throughput); and (2) loosing as little as possible from the advanced features offered by each resource (e.g. the versioning and ACL support of Amazon S3, or the super column families of Cassandra, and the vector clocks of Riak).

As a last goal, but by no means unimportant, we must guide ourselves by the principle that the users have different levels of knowledge and requirements, and thus granting them, if performance or other judicious reasons demands it, the possibility to jump over abstraction layers and gain access to the original API.

Our approach is to offer a software platform and a set of API as much as possible paradigm- and technology- independent. In order to face the challenges of this approach we model applications in terms of Cloud Building Blocks which are able to communicate each other.

A *Cloud Building Block* is any identifiable entity inside the cloud environment. Cloud Building Blocks can be cloud resources (which are under cloud provider control) or Cloud Components. A *Cloud Component* is a building block, controlled by user, configurable, exibiting a well defined behavior, implementing functionalities and exposing them to other application components, and whose instances run in a cloud environment consuming cloud resources. Simple examples of components are: a Java application runnable in a platform as a service environment; or a virtual machine, configured with its own operating system, its web server, its application server and a configured and customized e-commerce application on it. Components can be developed following any programming language, paradigm or *in-process* API. An instance of a cloud component is in a cloud environment what an instance of an object is in an object oriented application.

Communication between cloud components takes places through cloud resources (like message queues – AMPQ, or Amazon SQS) or through non-cloud resources (like socket-based applications). The same as component development, communication can use any paradigm and/or *remote* API.

Even if cloud components can be developed in many different way, the results are not equivalent in terms of the overall *quality* of the final application. A simple example is the case of the e-commerce application developed in a single virtual machine: even if it will be very easy to deploy a new e-commerce application and to improve the performance of the existing one changing the configuration of the virtual machine, the application does not scale well when the workload grows too much. The web application must be rewritten if there is a need of using more than one virtual machine.

Our project aims at offering a way to develop cloud components which are: (a) *scalable,* i.e. it should support multiple instances of the same component and scale well respect to the increasing number of instances; (b) *fault tolerant,* i.e. it should be able to handle in an automated as possible way the fault of component instances; (c) *manageable,* i.e. it should be easy to configure it, changing its working parameters; (d) *autonomous,* i.e. it should be able to run in an environment independently from other components.

The high level description of a cloud application, in terms of inter-communicating components, is not affected by the way in which components are developed (the programming language and the programming paradigm adopted) or communicate between them (like queues or sockets).

Achieving the aim of an unified API for multiple programming languages requires a specific architecture. We propose a layered architecture composed by: (1) native protocol; (2) native API; (3) driver API; (4) interoperability API; (5) connector API; (6) cloudlet API; (7) user components.

At the lowest layer we have either the *native resource protocol* (Web service, RPC, etc.), or a *native resource API* provided as a library by the vendor for a certain programming language; at this level we have no uniformity (for example nothing between CDMI resources and a Cassandra Thrift API), but we have full access to all the features of the resource, and no performance penalty.

One layer upwards we have the *driver API* which wraps the native API, providing the first level of uniformity: all resources of the same type are exported with the same interface. Thus exchanging, for example, an Amazon S3 with a Riak key-value store is just a matter of configuration. Of course at this level we are incurring some performance penalties (as we have to trans-code from native a data model to our uniform one), and we are starting to loose some custom resource features (like the ACL from Amazon S3).

The RPC-like-*interoperability API* aims to provide programming language interoperability and protocol syntax and semantic enforcements. It is not a full API, but actually an RPC solution that abstracts addressing, and provides the driver API with stubs, and the connector API proxies.

The first layer that the developer is expected to touch is the *connector API* which, depending on the programming language, provides abstractions for the cloud resources, suitable for the programming paradigm. It can be roughly compared with the C's ODBC, Java's JDBC, or even Java's JDO. In fact this is where we provide the second kind of uniformity for the programming paradigms, as all the implementations of the connector API in object oriented programming languages will have similar class hierarchies, method signatures, or patterns. About the performance only small impact is expected, and there is no feature loss due to the 1:1 mapping between what the driver and the connector API provide.

Even thought the developer already can access cloud resources, he or she must restrict himself or herself to a cloud compliant programming methodology, which we provide (integrated with all the layers already mentioned) that we call *Cloudlet*, as similar with the existing Java Servlet technology that provides standard programming components in J2EE environments and which was adopted even by Google in their AppEngine PaaS solution. Again like in the layer above little performance and no features are lost.

The native and driver APIs live inside a *driver daemon*, meanwhile the connector, cloudlet APIs, and the user code lives in what we call a component. Both processes are usually on the same node, but the resource does not have a clearly defined process scope, and in almost all cases it is outside of the current node. The communication between these three scopes is done as follows: between the driver daemon and the actual resource we use the native networked resource protocol (accessed through the native API), and between the the driver daemon and the component process we use our custom interoperability API (by providing stubs on the driver side, and proxies on the component side).

# 3 RELATED WORK

The requirements for cloud APIs only at the infrastructure level were recently synthesized in (OGF, 2010). A comprehensive study of the provider Web APIs is provided also in (Velte et al, 2009). The authors of (Mather et al, 2009) notice that API can manifest in different forms, and two types of APIs were identified: (1) APIs offered by IaaS cloud service providers allowing users to create and manage cloud resources, and (2) APIs that allow the description of common behaviors that apply to requests/responses, resource models which describe data structures used in requests/responses, as well as requests that may be sent to cloud resources, and the responses expected. We are dealing with the second type.

The Orleans programming model (Larus, 2010) is based also on the application decomposition in loosely coupled components, each of which executes in its own failure container. All communications between grains occurs across channels. We considered a more general case in which the components of our platform are communicating via resources (not necessary channels). The notion of building blocks for cloud computing and the assumption that the application developers are aware how to break the application into these building blocks is presented also in (Liu and Orban, 2008). The contexts are different: the paper that we mention is dealing with a library of operators for combining these building blocks, while we are talking about a more complex set of APIs.

The API architecture that we proposed is just a piece of mOSAIC, that encompass the development of a complete open-source platform allowing the development of applications using services from multiple cloud providers. Following the proposed timeframe of the development phase, a first proof-of-concept implementation of the full software stack behind the proposed cross platform API will be available in less than one year.

# REFERENCES


Larus, J. (2010). Programming Clouds. In *LNCS* 6011.

Liu, H., and Orban, D. (2008). GridBatch: Cloud Computing for Large-Scale Data-Intensive Batch Applications. In *CCGrid 2008*. IEEE Computer Press.

Mather, T., Kumaraswamy, S., Latif, S. (2009). *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance* O'Reilly Media.

OGF (2010). Open Cloud Computing Interface - Use cases and requirements for a Cloud API.

Velte, A.T, Velte, T.J. and Elsenpeter, R. (2009). *Cloud Computing, A Practical Approach*. McGraw-Hill.