An Approach for the Development of Portable Applications on PaaS Clouds

Filippo Giove, Davide Longoni, Majid Shokrolahi Yancheshmeh, Danilo Ardagna and Elisabetta Di Nitto
Politecnico di Milano, Milan, Italy

Keywords: Cloud Computing, Performance Study, Portability.

Abstract: Cloud computing is becoming important for ICT industry. Despite undoubted advantages in term of scalability and cost savings, today there are still issues regarding its massive diffusion due to portability of applications. To address this problem, in this paper we propose a new approach for the development of portable applications for Platform as a Service (PaaS) systems. This is based on a Java library exposing a *vendor independent* API that provides an abstract intermediation layer for the most important middleware services typically offered by PaaS systems (e.g., NoSQL services, message queues and memcache).
The current version of our library supports the portability of applications across Java platforms for *Google App Engine* and *Windows Azure*. We have conducted some experiments especially focusing on evaluating the performance degradation introduced by our library when executing an application on both PaaS. The experiments demonstrate that such degradation is not significant.

1 INTRODUCTION

Cloud computing is new paradigm which provides a new way of managing and offering IT services. Despite undoubted advantages of Cloud systems in term of scalability and cost saving, today there are some issues regarding limitation in their massive diffusion. Ideally, any company should be able to change its Cloud supplier. The reasons that can push an IT manager to move all or part of the company software from one supplier to another can be:

- offered SLA not respected;
- costs increase over time;
- competitors start new interesting or cheaper services.

Unfortunately, today moving from a Cloud provider to another is not an easy task. Each Cloud provider has its own API for deploying and running software and, most important, if we focus on the PaaS (Platform as a Service) level, each Cloud provider offers slightly different services, components and containers. This implies that an applications built for a certain PaaS and exploiting certain services (or components and containers) needs to be reengineered to be moved to a different Cloud. This is certainly a time consuming and very expensive operation that leads to the perception of so-called *Lock-in* by the Cloud service provider.

In this paper, we propose a library called CPIM (*Cloud Provider Independent Model*), that offers to developers PaaS-level services such as message queues, noSQL services, and caching service, abstracting from the details that are specific of the underlying PaaS provider. By exploiting our services an application developer is able to implement his application in a PaaS independent way. At deployment time, he/she specifies the actual PaaS to be used for the operation of the application and configures in the appropriate way the actual services to be used. At runtime, the CPIM library is in charge of acting as a mediator between the application code and the services offered by the PaaS that is currently hosting the application. Should the application owner be willing to move to a different Cloud, all he/she will have to do is to re-execute the deployment procedure on the other Cloud.

By exploiting the current implementation of the library, the application developer can build advanced distributed applications that, at the moment, can be deployed either on Google App Engine or on Windows Azure with no change at the code level. The deployment requires the execution of some specific configuration steps and is supported by an Eclipse plugin we have developed that guide the deployer through

Giove F., Longoni D., Shokrolahi Yancheshmeh M., Ardagna D. and Di Nitto E..
An Approach for the Development of Portable Applications on PaaS Clouds.
DOI: 10.5220/0004511605910601
In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science* (CI-2013), pages 591-601
ISBN: 978-989-8565-52-5

these steps.

The library has been validated by conducting experiments aiming at assessing the overhead it introduces at runtime while it acts as an intermediary between the application and the native PaaS APIs. The experiments have shown that such overhead is negligible.

The paper is structured as follows: Section 2 presents related work. Section 3 introduces the approach we have taken in building the CPIM library. Section 4 describes the application "Meeting in the Cloud" (MiC), we have developed and used as a reference for our experiments. Section 5 discusses the experimental tests we performed and the results we have obtained. Finally, Section 6 draws the conclusion and outlines future work.

2 RELATED WORK

The PaaS market is, in these days, rich of interesting initiatives. The first and most known solutions are those offered by Google App Engine (Euphrosine, 2013) and Windows Azure (Meier, 2013), but others are being launched and are gaining interest in the practitioners and scientific communities. Among the others, we can mention so called *open platform as a service* which allow to develop, test, and deploy applications implemented in any programming language, see for instance OpenShift (Red-Hat, 2012) and Cloud Foundry (CloudFoundry, 2013).

Currently each Cloud provider exposes services that, even when they are classified as similar, offer a different semantic and can be accessed through different APIs. This leads to serious problems of interoperability and portability of applications and of data residing on Cloud platforms.

There are some general approaches that could be taken to facilitate the portability across platforms (Petcu, 2011):

- Adoption of globally recognized and accepted standard APIs.
- Use of Wrapper/Adapter, i.e., a mediation layer that can decouple applications from the characteristics of Cloud platforms.

Among the standardization initiatives, IEEE P2301 (Ortiz, 2011) is a working group aiming to draw directives for the definition of Cloud services APIs. The goal is to define concepts, formats and conventions that, if widely adopted, would facilitate the portability and interoperability of different Clouds, allowing further growth and effective use of Cloud computing. Another example in this area is the partnership between Google and VMware (VMware and Google, 2012). The two companies, leaders of the Cloud market, through this alliance are trying to answer to portability issues by leveraging VMware vCloud technologies, Spring-Source, Google Web Toolkit and Google App Engine, to facilitate the development and implementation of business applications. Using the development environment based on Eclipse Spring Source Tool Suite, developers can create their own applications and later have the opportunity to carry out the deployment on a VMWare vSphere private environment, on a VMware vCloud partner or directly on Google AppEngine. OCCI, Open Cloud Computing Interface (OCCI, 2012) is another standardization activity that has been initially focusing on the creation of OCCI API for remote management of low level Clouds called IaaS (Infrastructure as a Service) and that now is trying to cover the PaaS model as well.

Among the mediation approaches, CSAL, Cloud Storage Abstraction Layer (Hill and Humphrey, 2010), provides an abstraction of typical storage services offered by most of PaaS, i.e., blobs and tables. The challenge of this project is related to the mapping of abstract operations in specific calls to such services.

SimpleCloud (Zend et al., 2013) is a open source project created by Zend Technologies in collaboration with IBM, Microsoft, Rackspace, Nirvanix and GoGrid, having the objective of improving the portability of PHP applications. With SimpleCloud, the developer can use a provider specific API or use a mediation layer that allows to write portable code among the supported set of vendors.

Among the current European-funded projects, mOSAIC (Open-Source API and Platform for Multiple Clouds) (mOSAIC, 2013) has the aim of developing an open-source PaaS as a mean of communication between Cloud applications and IaaS services. A mOSAIC application is able to postpone the choice of the underlying infrastructure. In other words, application developers and Cloud operators have the potential to defer decisions about which IaaS provider to use when the applications will actually run. The Cloud4SOA(Nikos Loutas, 2011) consortium seeks to consolidate three computing paradigms, namely Cloud computing, the service-oriented architecture (SOA) and lightweight semantics. The overall objective is to propose a reference architecture that allows interoperability between different Cloud vendors, facilitating the development, deployment and migration of applications among different Cloud PaaS providers. Specifically, Cloud4SOA adopts a semantic approach to allow developers and PaaS providers to express their technology-specific concepts with a

standardized vocabulary and common relationships.

Compared to the approaches presented above, CPIM library belongs to the category of mediation approaches. It differs from CSAL, since we provide an abstraction layer not only for services related to storage, but also for NoSQL, SQL, Queue, and Memcache services.

It also differs from all other approaches as it is the only one able to support both Windows Azure and Google App Engine which are very heterogeneous and can be considered the PaaS platforms market leaders. Among the other features, CPIM reconciles the strong difference in the semantics of the queue services offered by Azure and App Engine and implements two different services called *task queue*, adopting the App Engine semantics and *message queue* adopting the Azure semantics.

3 A LIBRARY FOR THE DEVELOPMENT OF PORTABLE PaaS APPLICATIONS

The most interesting characteristic of PaaSs is the set of abstractions they make available to application developers. Typical abstractions concern the concept of queue, which enables a reliable and asynchronous communication among distributed application components, and various different storage mechanisms ranging from a pure distributed file system to SQL and noSQL databases. All these abstractions are made available as services that can be invoked through well defined interfaces.

While the abstractions can be considered independent of each specific PaaS, the corresponding actual services are PaaS-specific and differ from PaaS to PaaS. The differences can either concern the interface they offer to programmers or their actual behavior. In both cases, these differences have a significant impact on the applications under development. In the following we present our solution to overcome these differences.

3.1 Solution Overview

The CPIM library aims at offering to application developers a homogeneous set of abstract PaaS services, that at runtime it adapts to the actual PaaS services being used. The services supported currently are the following:

- *SQL service* offers all capabilities of a traditional relational database and offers a SQL interface for interaction with its users.
- *Blob service* offers the possibility of storing simple bulk of data by relying on large storage resources offered by the Cloud.
- *NoSQL service* offers the capabilities of storing large quantities of data. Differently from SQL databases, these systems do not support completely the linguistic power of relational algebra as they do not offer the join operation, but they offer mechanisms to support high scalability and reliability.
- *TaskQueue service* offers the possibility to queue tasks to be executed and provides the logic layer that extracts tasks from the queue and executes them.
- *Message Queue service* offers the possibility to queue messages so that the sender and the recipient components can interact in a decoupled and asynchronous way.
- *Memcache service* offers the possibility to build an in-memory database up to a size of about 32 GB.
- *Mailing service* provides the operations to send an email message to some address specified by the user.

Figure 1 positions CPIM between an application and the two currently supported PaaS, Google AppEngine and Microsoft Azure). These last ones are relevant cases to study for two main reasons: first, they are the two market leaders at the moment, second, they offer quite specific and diversified services. Because of this second reason, we argue that demonstrating the feasibility of creating an abstraction layer on top of them allows us to show that the CPIM philosophy has the potential to work in general.

The left handside of Figure 1 highlights the presence of a configuration file through which the application developers provides information concerning the cloud to be adopted for deploying the application. This configuration file is not to be provided during the development of the application, which remains completely cloud agnostic, but should be available at the moment of deployment. Its specific content is described in detail in Section 3.5.

The design pattern used for implementing the li-

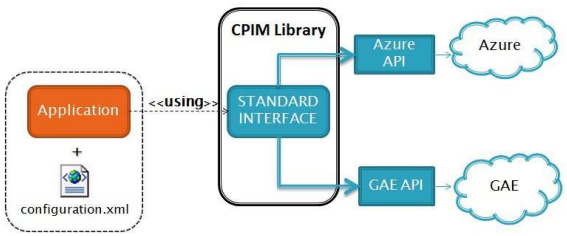

Figure 1: Overview of the CPIM library approach.

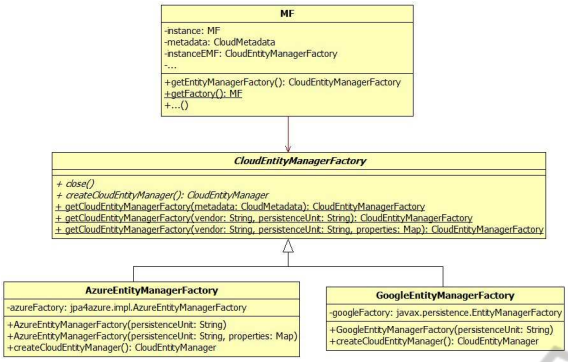

Figure 2: Usage of CPIM to initialize the NoSQL and Memcache services, GAE case.

brary is the *AbstractFactory* pattern (Metsker and Wake, 2006). More precisely, the developer of an application exploiting the library is supposed to exploit the primitives of a factory class to use all services offered by the library. Such a factory at runtime reads from the configuration file the information concerning the cloud provider to be used and instantiates the corresponding concrete factory that in turn, on request, instantiates the factory handling the specific service that the application is going to use. The sequence diagram shown in Figure 3 describes this interaction in the case the application has been deployed on Google App Engine where the NoSQL service is used.

For the sake of space we cannot provide details on all implemented services. In the following subsections we describe the NoSQL and Task Queue services as they have been the most challenging for the implementation of the library.

3.2 NoSQL Service

NoSQL services are provided natively by both App Engine, with its Datastore service, and Azure, with the Table Service. As discussed in (Hecht and Jablonski, 2011) (Han et al., 2011), a large number of NoSQL databases are available and support a variety of data models.

In order to cope with such variety, we have chosen to access such databases through a JPA (Java Persistence API) interface. JPA (Apache, 2012) (Hibernate,) is a specification defining a way to manage the persistency of objects on a database. As such, it allows us to abstract from the low-level APIs offered by the specific storage service, at the expenses of a reduced efficiency in the way queries and update operations are performed.

In case of Google App Engine, the interface and its implementation, are supplied directly by the SDK for Java Google App Engine. Vice versa, Microsoft does not offer an official JPA implementation for the Azure Table Service.

To compensate this limitation, we have adopted an open source library (jpa4azure Library,), and extended it by implementing the main methods for supporting the execution of queries. Thank to this extension, we standardize the invocation of queries written in SQL-like language.

The types of attributes remappable in properties supported by both platforms are: byte, date, Boolean, integer, String, Double. If the developer needs to insert an attribute of a type different from those supported, as long as serializable, this attribute should be defined as *@Embedded*. This annotation allows the JPA to convert the attribute into a byte array at the time of serialization. Vice versa, when the value of an attribute annotated with *@Embedded* is returned, the JPA converts the corresponding array in the object type declared in Java class.

3.3 Task Queue

A queue is a service frequently used by Web applications deployed in the Cloud. It allows to decouple the application tiers (e.g., the front-end and back-end) in a way that application containers supporting the runtime can scale in and scale out independently. Usually, queues are used to store messages that are transferred from the application front-end to the back-end or vice versa. Task queues offered by Google (Queue, 2013), instead, store tasks (or links to task code) that have to be activated as soon as the corresponding record is extracted from the queue. A task queue not only includes the mechanisms for storing information about the tasks, but also the mechanism that, typically adopting a FIFO ordering, gets from the queue a task and triggers its execution. This mechanism is particularly useful to support the execution of background operations.

Task queues appear to be useful in a variety of applications where it is possible to execute some computational intensive operations in the background. Thus, we have decided to offer them as part of the CPIM library and we have developed a software layer on top of the *Azure Queue Service* (AzureDOC, 2013) to implement them according to the schema in Figure 5.

In the CPIM library a task is represented as a *Cloud-independent* object. This object is represented by the *CloudTask* class, which encapsulates the following information (see Figure4):

- HTTP method used for the invocation of the task.
- Parameters passed to the task. These are represented by typical key-value pairs.
- Path of the task.
- Unique identifier of the task.

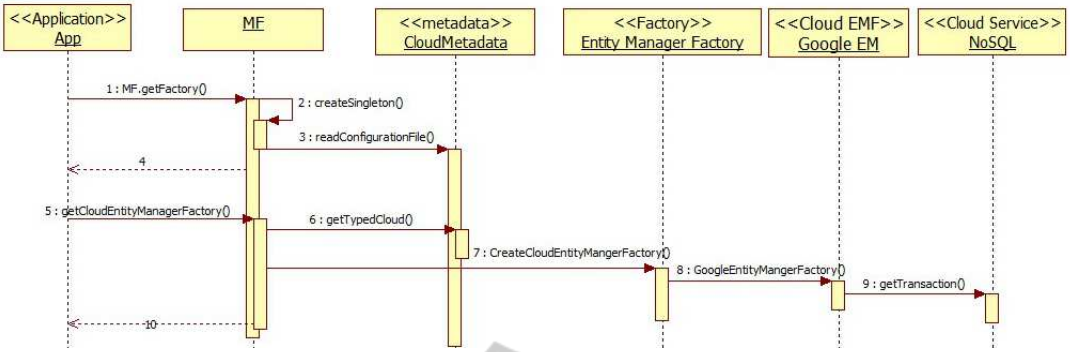

Figure 3: Sequence Diagram for MF and NoSQL.

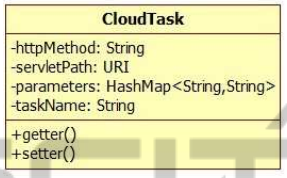

Figure 4: Class CloudTask.

In the case of Google, this information is extracted to populate the *TaskOption* object, which is then inserted into the App Engine *Push Queue*. Instead, for Azure, this information is extracted, properly formatted in a text message, and inserted in a *Message Queue*.

Azure requires also the implementation of an ad-hoc consumer executed in a dedicated Role to guarantee the separation between the presentation layer of the application from its back-end. In particular, in our implementation we instantiate a different consumer thread (exploiting the Azure *InternalWorker* class) for each task queue to monitor. This thread reads the messages from the queue and creates a *sub-consumer* dedicated to each read message (see Figure 5). Each sub-consumer:

- Extracts the information contained in the message.
- Invokes the task through the path included in the message.
- Deletes the message from the queue.

The consumer thread executes its operations periodically. The duration of such period can be customized by the user.

In the case of incorrect execution of a task, App Engine offers the possibility to automatically retry the execution of the same task. In Azure we rely on the fault tolerant features of Azure queues. Queues expect to obtain a feedback from the consumer of a message within a *Visibility Timeout* that by default is set to 10 minutes (AzureDOC, 2013) and put the message back into the queue in case this timeout has elapsed. Thanks to this mechanism, in case of failure of any component involved in the execution of the task, the

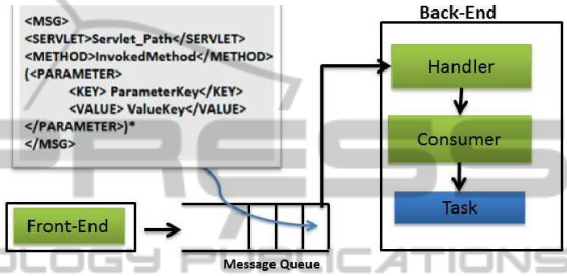

Figure 5: Implementation of Task Queue for Azure.

message providing information about the task will be read again and the task will be re-executed.

The sequence diagram in Figure 6 shows how our Azure task queue works. The diagram highlights the organization of software in two different parts, back-end and front-end that interact through the queue and, being decoupled, can be managed and scaled independently from each other depending on the actual load of each application part.

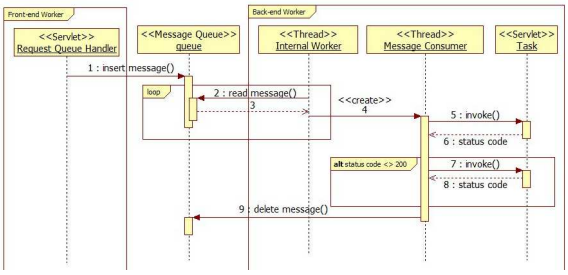

Figure 6: Sequence Diagram for the Amazon Task Queue

On the library side, the queue instantiation requires the configuration of various parameters, by using the *queue.xml* configuration file. This configuration file is read by the *CloudTaskQueueFactory* object that is then able to instantiate the *CloudTaskQueue* object (see Figure 7), the Cloud-independent wrapper used to communicate with the queues of the two providers. Once this object is obtained, it is possible to invoke the following methods for interacting with it:

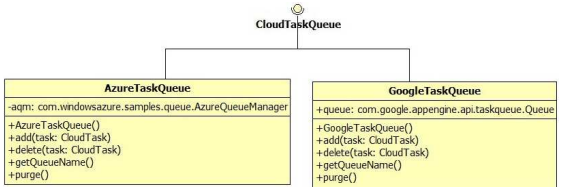

Figure 7: Class Diagram CloudTaskQueue.

- *add(CloudTask)*: allows to add a CloudTask to the queue
- *delete(CloudTask)*: allows to delete a CloudTask from the queue
- *purge()*: deletes all CloudTask
- *getQueueName()*: returns the name of the queue

3.4 Summary of CPIM Supported Services

Table 1 summarizes the characteristics and limits of the two PaaS and the extensions implemented in our library. The first column contains the abstract Cloud services, the second and the third columns are the Google App Engine and Azure solutions for each service, and the last column is the approach provided by our library. As it can be seen, the two platforms in some cases offer several solutions. When needed and in particular for the task queues, SQL and NoSQL services, we have built an extensive software layer to offer to the library users the richest possible semantics for the corresponding service.

3.5 Application Deployment

As discussed before, using the CPIM library application developers are able to build a Cloud-independent code even when they use PaaS level services. The next step is then to deploy the application on the selected PaaS. This, in general, is not a trivial task as each PaaS has its own procedure and configuration parameters to be set. For this reason, we have carefully defined the deployment steps for a CPIM-based application on the two supported PaaS and we have developed an Eclipse plugin to support this task.

Such plugin supports the deployer in the following tasks:

1. Specify the configuration files needed to identify the selected cloud and properly configure it.
2. Create an ad-hoc project including: *(i)* the package (typically jar files) of the vendor specific APIs, *(ii)* the package (war file) containing the application code and *(iii)* the CPIM library package (jar file).
3. In case task queues are used in Azure, prepare a separated deployment package (war) containing the implementation of the queue consumer.

The configuration file is a XML file which contains information such as choice of Cloud Provider and configuration for each services. For example, the *<vendor>* tag allows to set the Cloud platform to be used, while the *<sql>* tag contains the configuration of the SQL service that will be used for configuring JDBC. In Figure 8 a screenshot of the plugin is shown. The user can choose GAE and Azure and how he can enable the services required by his application.

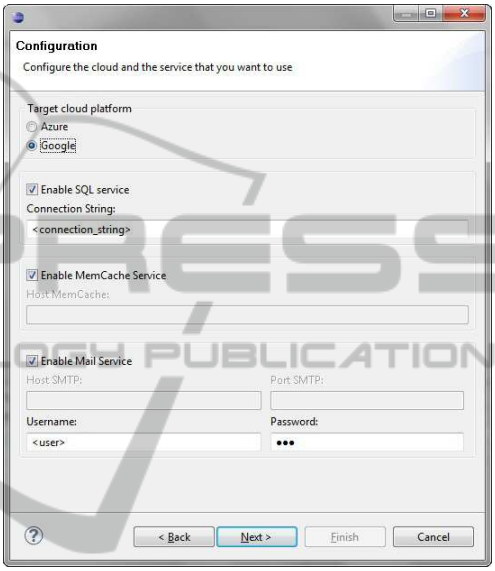

Figure 8: CPIM Plug in for eclipse.

If, for any reason, the need to change the Cloud provider arises, no re-engineering or modification of the code is required. Only simple changes in the configuration file are needed.

4 MEETING IN THE CLOUD APPLICATION

MiC (Meeting in the Cloud) is a social networking web application. It allows a user to register and to choose his/her topics of interest providing a grade in the range 1-5. At the end of the registration process, MiC identifies the most similar users in the social network according to the registered user's preferences, in particular, similarity is computed through the *Pearson coefficient* (Pearson, 2010). After registration, the user can enter into the MiC portal and can interact with his "Best Contacts" writing and reading posts on the selected topics.

Figure 9 shows the registration and user similarity computation processes by highlighting the application components that are involved in their execution.

Table 1: Summary of the services offered by App Engine, Azure and CPIM.

Service	Google App Engine	Windows Azure	Our Approach Library
NoSQL	Proprietary API JPA interface JDO interface	Proprietary API JPA interface(third-party)	JPA interface Support serializable objects (@Embedded)
SQL	MySQL-like Maximum 10GB for instance JDBC Driver	SQL Server-like sharing JDBC Driver	java.sql
Memcache	Synchronous/Asynchronous	Dedicated/Co-located Memcache supported by the Caching service(third-party)	Synchronous
Blob	Two typologies to manage Flat organization 32MB maximum blob size	Two blob typologies Hierarchical organization Max 200GB/1TB for Block/Page blob	Flat organization 32MB maximum blob size
Queue	Two typologies to manage code Task code	Message code Limits to 64KB for message	Message code Task Code Limits 64KB for message
Mail	Internal SMTP Direct API support java.mail	External SMTP support java.mail	support java.mail

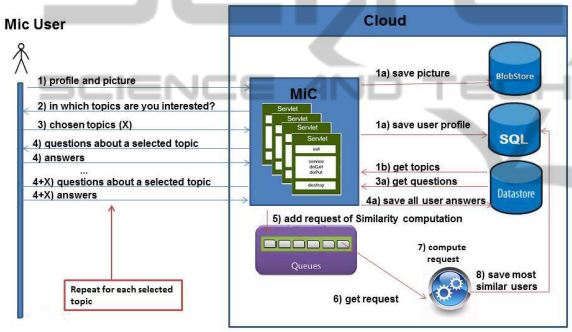

Figure 9: Registration and User Similarity computation of MiC.

More precisely, the application is composed of a front-end developed as JSPs and Servlets and a back-end developed as a CPIM *CloudTask*. The front-end and back-end are decoupled by a task queue and share user profiles through the SQL Service. This same service also stores messages, and best contacts that are accessed by the front-end. The Blob Service is used to store pictures while the NoSQL Service stores user interests and preferences. Both are accessed by the front-end. Finally, the Memcache Service is used to temporarily store the last retrieved user profiles and best contacts messages with the aim of improving the response time of the whole application.

Figures 10 and 11 show the component diagrams describing the deployment on Google App Engine and Microsoft Azure. In the case of Google App Engine, the components of the MiC software are:

- Web Application Project which contains:
  - MiC Component that includes the specific application-dependent Servlet classes and JSP pages as well as the CPIM API library and its configuration files
  - Google App Engine SDK for Java: SDK for the development of Java applications for GAE. It includes the Java API for accessing services offered by platform
  - Java SQL API: APIs to access and use relational DB
- GAE Services: services offered by Google

In the case of Microsoft Azure, Apache Tomcat has been deployed in each WorkerRole in order to use Java Servlet technology and the Component Diagram includes the following components:

- the same MiC Component as in the GAE case
- Internal Worker war: war containing the code for the queue manager we have developed for Azure.
- jpa4Azure API: API for accessing services Queue, Blob and Table (through the JPA interface)
- Java SQL API: API to access and use of relational databases
- SpyMemcache client: client component of the Memcache service
- Azure Services: Azure platform services

5 EXPERIMENTAL RESULTS

The objective of the evaluation has been to assess the overhead introduced by the adoption of our vendor independent. To do this, for each PaaS we deployed two implementations of the MiC application, one using our library and one using directly the vendor-specific API.

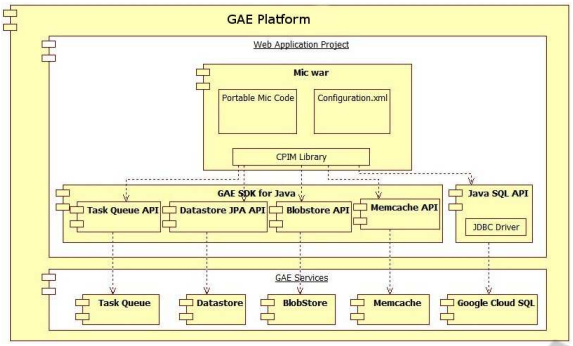

Figure 10: Component Diagram: deploy on Google App Engine.

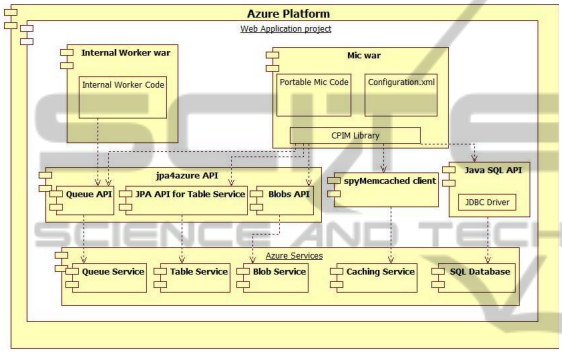

Figure 11: Component Diagram: deploy on Azure.

Experimental Setting. The metrics we have adopted for comparison are:

- The latency
- The computational overhead, in terms of CPU resource utilization

We used JMeter (Apache, 2013) as workload injector and we considered the following two workload scenarios as they cover all actions a user can perform on MiC:

Test Plan 1. The user session registration as described in Figure 9.

Test Plan 2. A session where the user performs the following actions:

- Login
- Update profile
- Write a message
- Logout

The content of the storage system was the same for all tests:

- SQL Database:
  - Number of Tuples in table Message: 3,000
  - Number of Tuples in table User Profile: 4,300
  - Number of Tuples in table User Similarity: 14,800
- NoSQL Table Service:
  - Number of entities for User Ratings: 14,800
  - Number of entities for Topic: 7

On Azure the tests were carried out deploying MiC on two Worker Roles (one for the front-end and one for the back-end) by using Small instances, while for GAE the tests were carried out deploying MiC on two instances with F1 size. Tables 2 and 3 show, for each request, the cloud services that have been used, the concerned tier and the think time introduced to make realistic behavior of a user in the session. More specifically, we assume that the think time follows a Gaussian distribution for which we provide in the table the average (parameter K) and the standard deviation (parameter D).

Table 2: Summary Table Test Plan 1.

HTTP REQUEST	Used cloud services	Think Time(sec)
REGISTER	Blob, SQL	(K=10;D=2)
SELECT TOPICS	-	(K=5;D=1)
SAVE ANSWER	NoSQL, TaskQueue	(K=20;) (D=5)

Table 3: Summary Table Test Plan 2.

HTTP REQUEST	Used cloud services	Think Time(sec)
LOGIN	SQL,Memcache,NoSQL,Blob	(K=5;)(D=1)
EDIT PROFILE	-	(K=2;) (D=0.1)
SELECT TOPICS	NoSQL,SQL,Memcache	(K=5;) (D=1)
SAVE ANSWER	NoSQL,TaskQueue	(K=20;)(D=5)
REFRESH	NoSQL,SQL,Memcache	(K=2;)(D=0.5)
WRITE POST	SQL	(K=10;)(D=3)
LOGOUT	Memcache	-

Latency measures have been collected directly through JMeter while to capture data regarding the CPU utilization of the Azure and Google App Engine containers we had to use some platform specific tools.

Azure offers the ability to remotely access the VMhost of the application deployment (using Remote Desktop Protocol). Thanks to this, we collected data directly from the monitoring tool of Azure for the CPU utilization. App Engine does not offer the possibility to access to virtual machine-level information. Thus, we had to exploit the Quota service API by instrumenting the code of MiC. In this way, through the Java SDK, we were able to get for specific blocks of code, the number of CPU cycles consumed by executing the block within the sandbox App Engine.

The unit of measure used by the API is *machine megacycles*. If all instructions are performed sequentially on a reference machine 1.2 GHz 64-bit x86 CPU, 1200 megacycles are equivalent to one CPU second devoted for a block execution.

Using the obtained data, we can estimate the percentage of use of the CPU, through the following formula:

$$\frac{\sum_{k=1}^{N}(M_k)}{1200*T}$$

where M_k indicates the number of megacycles consumed by the k-th request within a test, N denotes the total number of requests made during a test, while T indicates the duration in seconds of the whole test.

Results on Azure. Figure 12 reports the latency time results for Azure in the two test scenarios when the number of users grows from 1 to 50. In this way we were able to vary the VM role utilization between 2 and 70 %, analysing the behaviour of the system both in light and medium/heavy load conditions. As it appears from the graphs, no significant difference between the use of our library and the native Azure APIs can be observed.

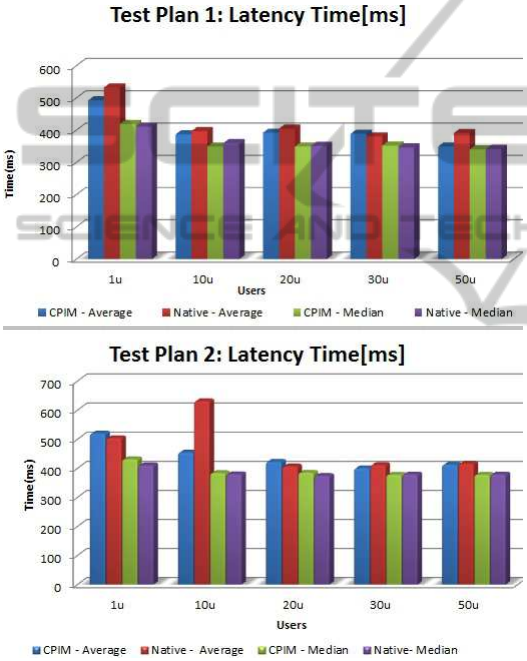

Figure 12: Latency Time Evaluation Test: CPIM results vs Azure Native results for Test 1 and 2.

A careful analysis was done for the results obtained in the second test plan with 10 users, where a higher average latency for the native version has been observed. Specifically, the results of the queries LOGIN, SELECT TOPICS and SAVE ANSWERS presented some differences greater than 300 milliseconds. The Response time graph in Figure 13 shows that requests with large latency times (around 30 seconds) occurred during three time intervals. We guess that these large delays have been caused by the migration of the VM which hosts the application on a different machine. This caused a temporary block in requests execution. This problem, as shown in the graph of Figure 14, has not occurred in the test case using our library where the maximum registered response time has been below 7 seconds.

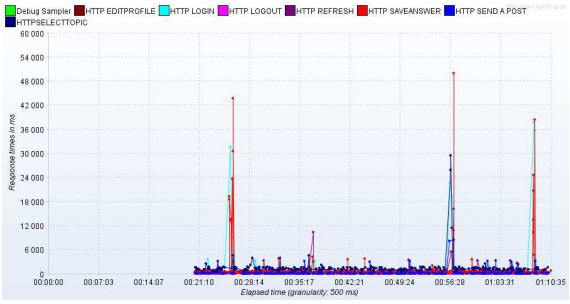

Figure 13: Response Time - 10 users version Native.

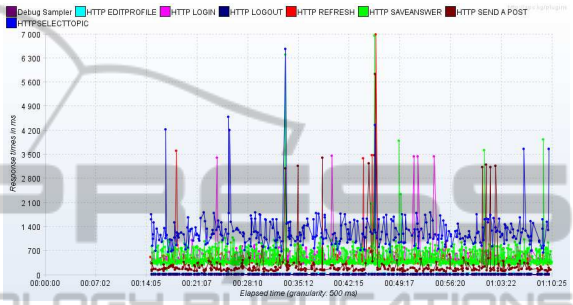

Figure 14: Response Time - 10 users version by CPIM.

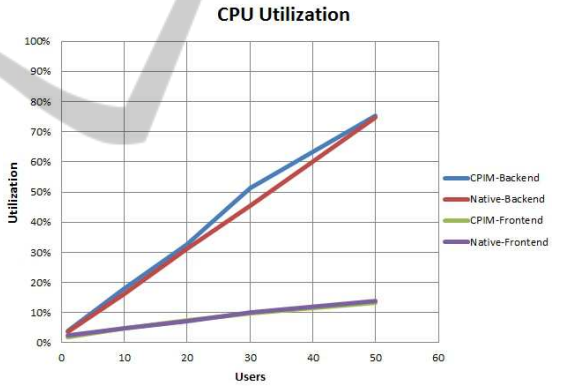

Figure 15: CPU utilization Test: CPIM results vs Azure Native results.

Figure 15 reports the CPU utilization gathered during the tests. The computational overhead introduced by our library both for the front-end and back-end is almost negligible, even under heavy load (the green line representing the front-end CPU utilization in the CPIM case is completely hidden by the line representing the utilization in the native case). This result confirms that our implementation of the Task Queue service, which resides in the back-end, has good performance.

Results on Google App Engine. Similar results have been obtained for the Google platform. These are reported in Figure 16 for latency time and in Figure17 for CPU utilization.

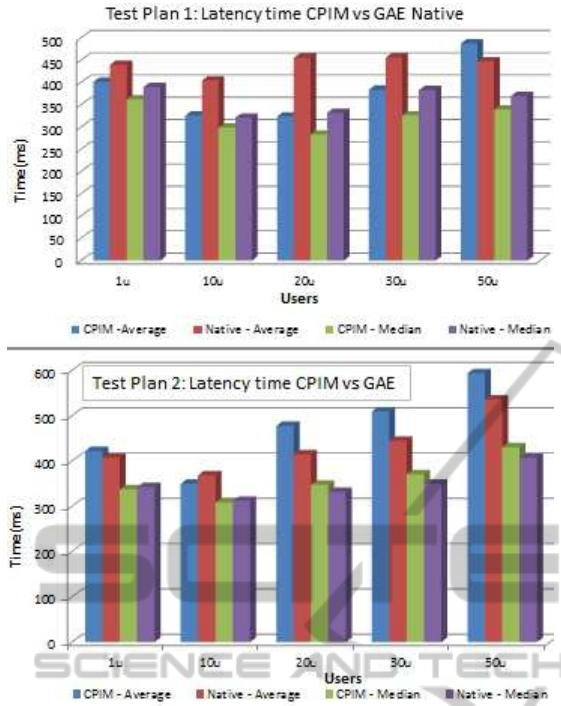

Figure 16: Latency Time Evaluation Test: CPIM results vs GAE Native results for Test 1 and 2.

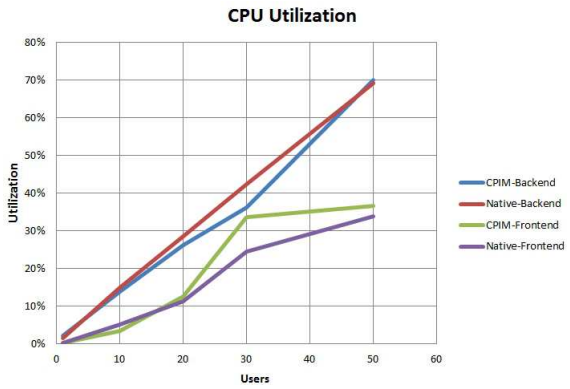

Figure 17: CPU utilization Test: CPIM results vs GAE Native results.

6 CONCLUSIONS AND FUTURE WORK

Avoiding vendor lock-in is an important issue that becomes quite difficult to achieve when exploiting PaaS.

In this paper we have proposed the development of an abstraction layer exposing a "vendor-independent" API, which allows developers to use the most important and common services offered by PaaS systems. The approach is oriented to the Java language and the platforms currently supported are Google App Engine and Windows Azure. Using a social networking application as a test case, we have carried out tests to evaluate the overhead introduced by the use of the abstraction layer with respect to the direct use of the proprietary APIs provided by the two vendors. The results have shown that the overhead introduced by the library, both for the latency and CPU utilization, is negligible.

Future work will consider the integration of additional services offered by the two platforms. Furthermore, additional Cloud systems both at the PaaS and IaaS layers will be supported.

ACKNOWLEDGEMENTS

This research has been partially supported by the European Commission, Grant no. FP7-ICT-2011-8-318484 MODAClouds project (MODAClouds, 2013).

REFERENCES

Apache (2012). Java Data Objects (JDO API 3.0). http://db.apache.org/jdo/jdo_v_jpa.html.

Apache (2013). *Jmeter USer Manual Document*. Apache.

AzureDOC (2013). http://msdn.microsoft.com/en-us/library/windowsazure/hh767287.aspx.

CloudFoundry (2013). CloudFoundy. http://www.cloudfoundry.com/about.

Euphrosine, J. (2013). *Getting Started with Google Tasks API on Google App Engine*.

Han, J., Haihong, E., Le, G., and Du, J. (2011). Survey on nosql database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363 –366.

Hecht, R. and Jablonski, S. (2011). Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pages 336 –341.

Hibernate. *Hibernate Developer Guide*. Hibernate.

Hill, Z. and Humphrey, M. (2010). CSAL: A Cloud Storage Abstraction Layer to Enable Portable Cloud Applications. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 504 –511.

jpa4azure Library. JAR jpa4azure.jar (version 0.7). http://jpa4azure.codeplex.com/.

Meier, J. (2013). *Windows Azure Developer Guidance Map*.

Metsker, S. J. and Wake, W. C. (2006). *Design Patterns in Java*, chapter 17. Addison-Wesley Professional.

MODAClouds (2013). http://www.modaclouds.eu/.

mOSAIC (2013). http://www.mosaic-cloud.eu/.

Nikos Loutas, C. (2011). *Cloud4SOA Reference Architecture*.

OCCI (2012). http://occi-wg.org/.

Ortiz, S. (2011). The problem with cloud-computing standardization. *Computer*, 44(7):14 –15.

Pearson, K. (2010). Similarity Metrics: Pearson Correlation Coefficient. http://mines.humanoriented.com/classes/2010/fall/csci568/portfolio_exports/sphilip/ pear.html.

Petcu, D. (2011). Portability and interoperability between clouds: Challenges and case study. In Abramowicz, W., Llorente, I., Surridge, M., Zisman, A., and Vayssire, J., editors, *Towards a Service-Based Internet*, volume 6994 of *Lecture Notes in Computer Science*, pages 62–74. Springer Berlin Heidelberg.

Queue, G. T. (2013). Task Queue Java API Overview. https://developers.google.com/appengine/docs/java/taskqueue/overview?hl=en.

Red-Hat (2012). *OpenShift - User Guide : Using OpenShift to manage your applications in the cloud*, edition 2.0 edition.

VMware and Google (2012). www.vmware.com/cloudportability.

Zend, IBM, Microsoft, Rackspace, Nirvanix, and GoGrid (2013). SimpleCloud API. http://www.simplecloud.org/.