An Open Framework to Support the Development of Commercial Cloud Offerings based on Pre-Existing Applications

Alexander Lenk
FZI Karlsruhe
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
lenk@fzi.de

Jens Nimis
FZI Karlsruhe
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
nimis@fzi.de

Thomas Sandholm
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
thomas.e.sandholm@hp.com

Stefan Tai
FZI Karlsruhe
Haid-und-Neu-Str. 10-14
76131 Karlsruhe, Germany
tai@fzi.de

ABSTRACT
Web-based service delivery and billing by consumption are two defining properties of Cloud Computing. They leverage novel business models and sales channels for yet to develop as well as pre-existing applications. Many pre-existing applications are already capable of running in distributed environments, but still do not meet the requirements to run as a Cloud offering. For example, applications built to run in cluster environments are designed for distribution and massive scalability and from an architectural point of view qualify to be suitable for Cloud environments as well. However, a limitation of cluster-based applications with respect to Cloud adoption is that many of them initially do not support necessary Cloud service features such as payment services or multi-tenancy. To close this gap, we propose an open framework that helps to adapt pre-existing applications into commercial Cloud offerings. The framework facilitates the process to extend pre-existing applications with the respective required web service interfaces, which ultimately allows them to be consumed as Cloud offerings in a pay-as-you-go manner. The framework approach is illustrated by the transformation of the two well-known cluster applications Hadoop MapReduce and MySQL into full-blown Cloud offerings.

Keywords
Cloud Computing, Framework, Hadoop, MySQL

1. INTRODUCTION
The pervasiveness of the Internet allows for Web-based applications to address a virtually unlimited audience. The success of such applications measured in page hits and unique users poses an enormous challenge towards application design with respect to scalability. Even an unexpected spike in the application popularity must not lead to a degradation of service quality - to every user the application must feel like he would have exclusive access.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCV Conference 2010, May 17–18, 2010, Singapore.
Copyright 2010 CCV

In recent years a large number of cluster applications have been developed, which have the ability to scale well for heavy workloads but which are not necessarily suited for multiple concurrent users or for Web access.

In this respects, such applications are not ready for the Cloud. To facilitate and accelerate the transformation that is needed to put them into the Cloud is the main goal of our framework presented in this paper. The framework follows a lightweight approach that tries to avoid invasive changes to the pre-existing applications and instead builds a system of loosely coupled services around it.

Section 2 presents the design of the framework and shows how payment and authentication services are integrated with pre-existing applications via a service mashup. In Section 3 the usage of the framework is illustrated for the Hadoop MapReduce and MySQL applications and in doing so the suitability and reusability are underlined. Section 4 discusses related work and Section 5 concludes the paper.

2. FRAMEWORK DESIGN
As shown in Figure 1 we choose layering as the fundamental architectural style for our framework. The constituent layers and their components comply with our Cloud computing stack [12]. They are described in detail in the following subsections.

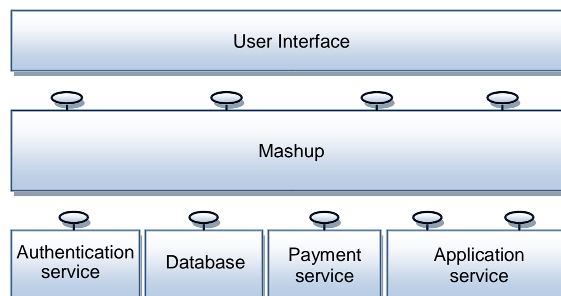

Figure 1: General framework overview

2.1 Identification of main components

In order to interact with the user, the framework needs a user interface which is located in the *Application* layer of the stack (see Figure 2). The user interface provides an easy access to the services located in the *Application Services* layer. Due to the fact that several services should be combined to one higher service, an additional component in the *Composite Application Services* Layer of the stack is necessary. We call this component *mashup*. The mashup uses different *Basic Services* such as a basic application service (further called *application service*), an authentication service, a payment service and a database.

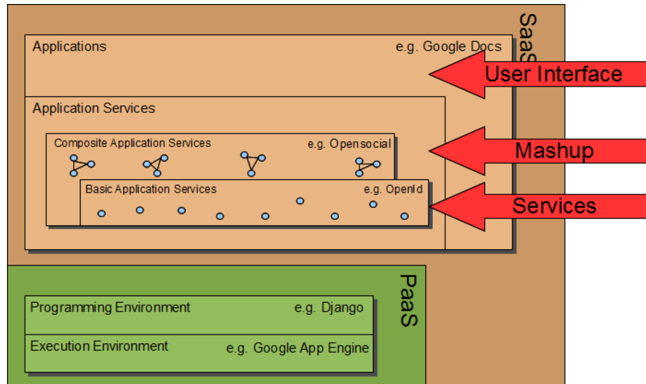

Figure 2: Identification of the components in the Cloud computing stack [12]

2.2 Application Service

The transformation into an application service is the first step of turning an existing cluster application into a cloud service. Therefore, the application service adds a payment and control interface to the cluster application. The service that provides both Web services is structured as a nested framework, linking the cluster application with a billing component and billing database (see Figure 3).The cluster application has to meet certain preconditions, such as it must be possible to control it externally by using the command line, RPC calls or Web service calls. It also allows the separation of its resources on user level. This could mean that user management is included in the application, which provides basic authentication and authorization functionality.

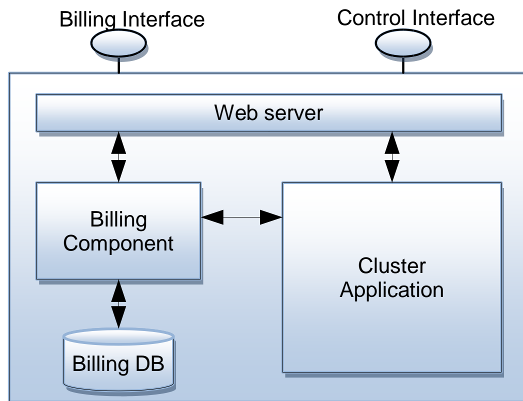

Figure 3: Application service overview

Because the application is not yet an application service (according to our definition) it has most likely no native payment or control Web service interface. The billing component is used to provide the functionality of the payment interface. It tracks the usage of the application and thereby deducts the budgets of the user and saves all the information in a billing database inside the application service. The algorithm which is used to deduct the budget could be a simple time based, or a complex one like a market based algorithm. The billing component is accessed by a Web service interface and can control the application if a certain event occurs (e.g. the user run out of funds the component makes sure the user can't perform any further actions on the application). We call the interfaces of the application service *Billing Interface* and *Control Interface*.

2.2.1 Billing Interface

The payment interface is used for all monetary transactions with the application service. The methods for the payment interface are *config*, *spending* (for market [18] or service level [4] based mechanisms), *deposit*, *budget*, *invoice,* and *info*.

Billing/config requires no input and outputs the current configuration. A configuration could, for example, contain information like the minimum amount the user has to pay, or what service level is guaranteed. Based on this information, the user can decide whether to use the service and how much to pay. This function is public and thereby also visible through the mashup.

Billing/spending is used by the user to set his willingness to pay. Based on this value a market mechanism could decide, what resource share has to be reserved for the user. The spending rate is also used by the budget tool to determine how much money will be deducted from the account. The input of this public method is the identification of the user and his new spending rate.

Billing/deposit can be used by the payment module to change the available budget on the application service. It has to be ensured that the user of the cloud service can't access this method. The input of the function is the identification of the user and the amount which is added to the budget (negative amounts get deducted).

Billing/budget is used to return the available budget on the application service.

Billing/invoice requires the dates for the invoice and returns the billing information for the given time.

Billing/info can output information like the actual share the user has in a market driven approach.

2.2.2 Control Interface

The control interface provides an interface to the cluster application's core functionality. E.g. for a database application a method for executing queries on the database is available. If the application is a file storage solution one method is to show the user all the files located on the storage. It is important to make sure that only the currently authenticated user is able to see his private data. This is done by passing the user identification to the control interface in every method call.

2.3 Payment Service

We propose to use an external, well-established payment provider to handle the payment process. It is also possible to implement the whole payment process in the payment module but there are plenty different payment providers, which are well known and have the trust of many users (e.g. Paypal [17]). Using an existing one lowers the barrier for new users and reduces the implementation effort.

2.4 Authentication Service

To lower the barrier for new users to use the service we recommend to provide a login using an external Shared Sign-On provider such as OpenID [11]. In the example of OpenID every customer who has already an OpenID identifier (e.g. a Yahoo [23] email address) can use the service without going through a complex registration process. If necessary only some additional information can be asked when the ID is used the first time. Some Shared Sign-On providers have the handicap that a browser has to be used to sign in. In that case we recommend having an additional provider allowing authentication without the use of a browser. Otherwise is it not possible to access the Web services interfaces from inside other programs.

2.5 Database

The database saves the state of the mashup. Its primarily used by the user module to save the identification of the user and his additional information. Other modules are also able to save data in the database. For instance the payment module to log the transactions or the authentication module can save access statistics. The database could be a conventional database like MySQL [7] or a cloud based one such as Amazons SimpleDB [1].

2.6 Mashup

The mashup is the central component of the framework connecting the underlying services and providing their functionality to the layers above. The mashup and the services are loosely coupled and can run on different machines, so for each underlying service there must be at least one *module* that communicates with the service. The mashup consists of the following modules: application module, payment module, authentication module, user module, and database module (see Figure 4).

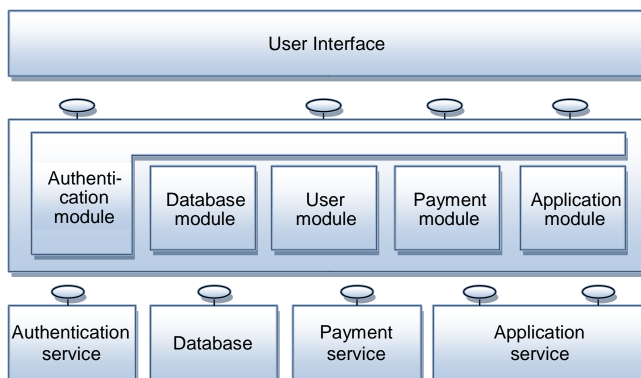

Figure 4: Detailed framework overview

2.6.1 Application Module

The application module is responsible for handling all the communication with the application service. This means it provides interfaces to the application service, the user, and other modules. The module gives the user access to public methods and hides the private ones. An example for an private method is the deposit method. Only the payment module is allowed to access this method directly so it is located in the internal, private interface. Otherwise every user could deposit money for free. The application module also takes care of passing the correct authentication token to the application service ensuring the user has only access to his data. Figure 5 shows a sample call on a storage application service.

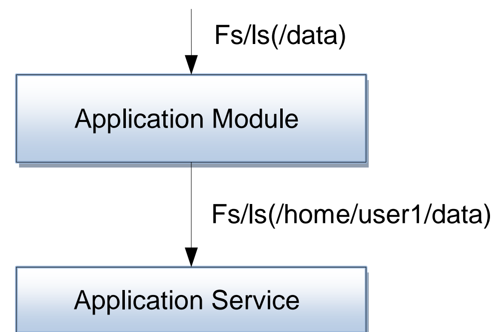

Figure 5: Forwarding a Web service call to the application service

Considering the case that the user wants to access the method *ls* at the storage application service. The application module gets the information about the currently authenticated user from the authentication module and adds the user id to the path before forwarding the request to the Web service of the application service. This ensures that a certain user has only access to his personal folder.

2.6.2 Payment Module

The user can use the payment module to deposit money on the application service account. The application service uses the deposit money then to determine how much resources the user can consume. The whole billing process is handled in the application service: the payment module just adds money to the account. Due to the fact that the payment process is performed by an external entity, the payment module is responsible for the transactions between the payment service and the application service. It ensures the payment process with the external service provider is complete before passing all the necessary payment data to the application module. The communication with the payment service could work in two different ways: the Cloud Service could check in constant periods if there were new payments (pull strategy) or the payment service could notify the cloud module if a transaction is done (push strategy). We propose that the push strategy is used if the payment provider supports it. Figure 6 shows the sequence of the payment process with a push strategy.

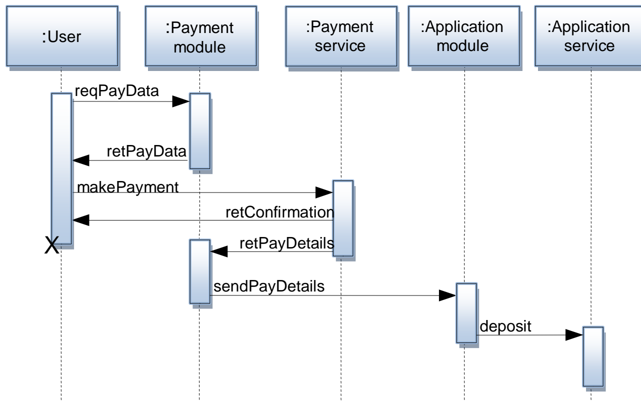

Figure 6: Sequence of the payment process

The user first requests all available payment services and the necessary data to perform the payment (*reqPayData*). After receiving this information he chooses one provider and uses the provided information to perform the payment (*makePayment*). He receives a confirmation from the payment provider and the provider also notifies the payment module that the payment was successful (*retPayDetails*). The payment module passes the payment details to the application module which deposits the payment to the user account in the application service.

2.6.3 Authentication Module

The framework supports several different types of authentication systems. The authentication module gives access to the various systems. The system could be an external authentication service, like OpenID, which needs a Web browser to perform the authentication task; the system could also be an internal authentication algorithm, like signing the query with a secret key. The data which is used for the authentication methods are stored in the database. The authentication module provides the method *request authentication types* to the user. The method is used to request all the information from the module for every single authentication system. For a shared authentication system, like OpenID, the method returns the login URL for OpenID. Another method, provided by the module, is *check validity*. This method checks if the authentication data is still valid, returning a Boolean which indicates the current status of the authentication. The authentication module is the core module of the mashup. Every request to the mashup must pass the authentication layer, ensuring no unauthenticated users have access to the other modules. Figure 7 shows the authentication process when accessing the application module.

Before communicating with the application module the user requests all available authentication types from the authentication module (*reqAuthTypes*). After choosing a certain type he uses the information provided by the module to contact the authentication service and process the authentication process with this external provider (*doAuth*). Finally the user accesses the application module providing the authentication data as well as the data necessary by the application module. The authentication module preprocesses the request filtering the authentication data out and checking the provided information with the authentication service. If the data is valid it forwards the request to the application module.

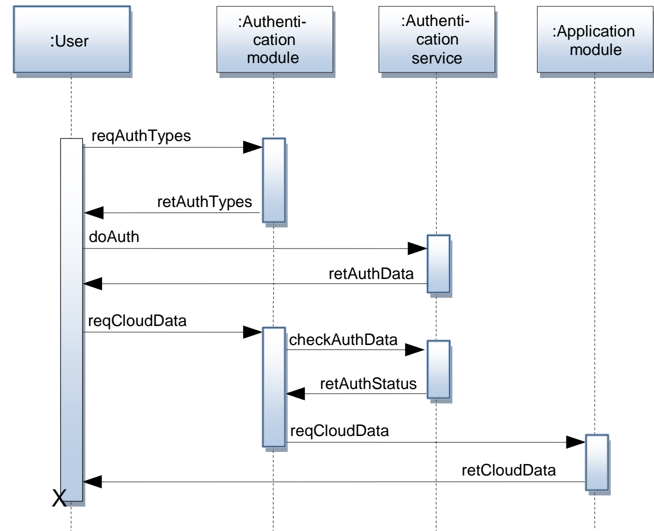

Figure 7: Sequence of the authentication process with an external authentication provider

2.6.4 User Module

To make sure the user can be identified in the case of a fraud, an accounting system is required. The user module implements this accounting system. When a new user is added to the framework, the user module assigns an unique ID and saves information such as first name, mailing address, or email address. The information provided by the user could be checked using external providers to verify their validity.

2.6.5 Database Module

The database module is used to give other modules access to the persistent storage, the database. Every module is able to store data in the database but no interface is provided to the user directly.

2.7 User Interface

The user could use all the functionality of the service through the Web service interface of the mashup. The mashup uses standard Web service technologies like REST [16] or implements a HTTP GET Web service interface and could be accessed using a browser. To make the interface more usable for an inexperienced user, a graphical interface or command line interface is proposed. The user interface connects to the Web service interfaces of the mashup to provide its functionality. The UI is nothing else than just a other way of accessing the functions provided by the mashup. The UI can run on a different machine than the mashup. It could be another Web server in case of an HTML/AJAX [10] frontend or the user's home machine in case of a command line interface.

3. PROOF OF CONCEPT

To proof the concept the usage of the framework is demonstrated with the cluster applications Hadoop MapReduce and MySQL.

3.1 Hadoop MapReduce

The MapReduce module of the Apache project Hadoop is a software framework to make it easier to develop and run applications that process vast amounts of data (multi-terabyte data-sets) in parallel on large clusters. Even if the clusters are comprised of thousands of nodes of unreliable hardware, Hadoop processes the data in a reliable, fault tolerant manner. The MapReduce software splits the input data into independent chunks and processes them in the map phase completely in parallel. Each map job processes one chunk of the input data. In the reduce phase the output of the map jobs are processed by one or more reduce jobs. Both the map and the reduce algorithms are written by the user of the cluster. Together the map and reduce algorithms are called a MapReduce program. [22]

Our goal is to design a Cloud service of Hadoop MapReduce that can be accessed by humans as well as machines (e.g. scripts). It should be possible to quickly gain access to resources for a limited amount of time. The barrier for using the service should be as low as possible, thus the payment and authentication services have to be well known and trusted.

3.1.1 General Setup

MapReduce has two main nodes that control the whole cluster: the Jobtracker and the Namenode. The Jobtracker controls the map and reduce processes and the Namenode the distributed Hadoop file system (HDFS). Within the Jobtracker the scheduler is responsible for allocating the different map and reduce jobs to the working nodes. Failed and incomplete jobs are also handled by the Jobtracker. In order to mash the Hadoop MapReduce cluster application with the other services and thereby transform it to the final Cloud service, Hadoop MapReduce has first to be transformed into an application service. In the second step it has to be decided what authentication and payment methods should be used. So in the third step the mashup and the user interface can be implemented.

3.1.2 Application Service

MapReduce has no native billing or control interface in order to achieve the needed functionality, so both interfaces have to be added. The MapReduce software already has a native Web server (Servlet) functionality built in that can be used from a scheduler plug-in .

The key enabler of running Hadoop as a service is the Dynamic Priority Scheduler, which was added in Hadoop 0.21. It enforces cluster shares in real-time based on bids from users, and allocates shares based on spending rates and budgets.

In this work the scheduler was extended to use an external database to get information on how much the users are willing to pay and to thereby determine the resource capacity share of each user. By default the scheduler uses simple file storage to persist budgets and relies on a REST interface to control the budgets and spending rates.

In figure 8 the architecture of the application service is shown.

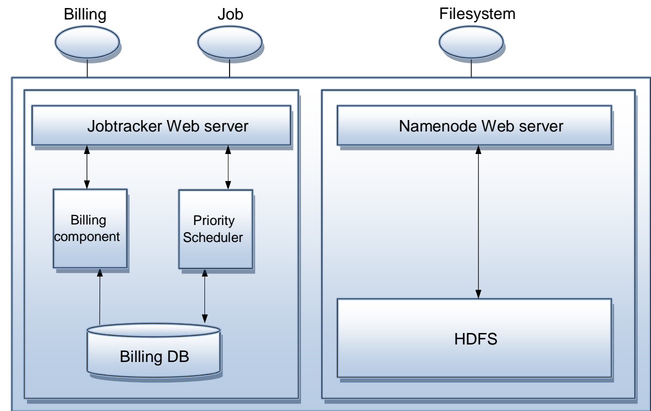

Figure 8: The Hadoop MapReduce application service

While the implementation of the control interface is straight forward and everything can be implemented in servlets running on the Namenode and Jobtracker, the billing component providing the billing interface is more complex.

3.1.2.1 Control Interface

MapReduce usually has two discrete nodes for controlling the jobs and controlling the file system. So there will be two different endpoints: the *Job* Web service and the *Filesystem* Web service.

Functions of the file system interface are *delete*, *list*, *create*, *upload*, and *download*. All functions require a HDFS URI [22] as parameter and return XML or binary data.

Filesystem/delete deletes a file or folder

Filesystem /list lists the content of a folder

Filesystem /create creates a folder

Filesystem /upload loads a file to the file system

Filesystem/download gets a file as binary data from the file system

The functions of the job Web service are info, start, and stop. Each function requires the queue name as user id as parameter and returns XML data.

Job/info returns the configuration of the Jobtracker, all the jobs and the share of the user.

Job/start creates a new Hadoop MapReduce job. As additional parameters it requires the job configuration as XML, the URI to the MapReduce program, and the HDFS URI to both input and output.

Job/stop requires the job id as input and stops the corresponding job on the cluster.

3.1.2.2 Billing Interface

The billing component is a standalone java tool running on the Jobtracker and deducting the budget of each user with active jobs after a certain amount of time. The deduction algorithm is by default included in the scheduler, but in this work an external component was used to minimize the overhead of the internal scheduler. Due to the fact that the priority scheduler is using an external database it is easy to modify the data the scheduler accesses.

The billing interface implements the functions *config*, *deposit*, *spending, budget, invoice,* and *info*. Each function requires the queue name of the user.

Billing/config information about the billing interval (the time the spending rate gets deducted from the budget), the available nodes and the SLA of the cluster.

Billing/deposit adds or reduces the amount of available funds on the users account. It requires the amount to be added or reduced (positive values get added, negative values get reduced).

Billing/spending sets the new spending rate of the user.

Billing/budget is used to return the available budget on the application service.

Billing/invoice requires the dates for the invoice and returns the billing information for the given time.

Billing/info outputs the current spending rate and share of resources that are reserved for the user with his current spending rate.

3.1.2.3 Billing component

The billing component takes care of deducting the current spending of the user accounts. It checks if there are running jobs on the cluster and in that case reduces the amount according to the current spending rate. It also recalculates the share each user has which is used by the scheduler to reserve the resources for the different queues. If the user is running out of funds it sets the spending rate and the users share to null.

3.1.3 Authentication Service

To achieve the goal that each user has an easy way of authenticating against the service the service provider OpenID is used as an external authentication provider. Unfortunately OpenID requires a Web browser to work, so for an authentication using a direct Web services access another authentication mechanism is needed.

3.1.3.1 Browser Authentication

OpenID is a decentralized network of providers that are able to create accounts. The providers are responsible for ensuring that the provided data of each user is valid. In the authentication process the user authenticates himself against the OpenID provider, telling him which website he wants to visit. The provider redirects the user to the provided web site and gives the web site an authentication token that authenticates the user. Usually this token is stored as a cookie in the user's browser.

3.1.3.2 Direct Authentication

After creating a user account and logging in the first time using a browser a secret key is generated and provided to the user. The user can sign the calls to the cloud service with this secret key and can thereby authenticate herself against the service.

3.1.4 Payment Service

The external payment service could be any service available on the internet that provides a Web service API. For the Hadoop MapReduce cloud service the provider Paypal is chosen because it is widely known, available in many countries, trusted, and provides the functionality needed to use the push strategy. The sequence of a call on the payment service is described in subsection 2.6.2.

3.1.5 Database

The purpose of the database is to store information of the users, like their authentication information and billing information. It could be any existing database which is supported by the language the mashup uses. We chose MySQL here because many libraries in many programming languages support this type of database. Further, it can be used as a cluster application and thereby be transformed into a Cloud service as well (see 3.2).

3.1.6 Mashup

As the core unit of the cloud service, the mashup hides and connects all the interfaces of the underlying services and applications and provides public interfaces to the user. It can be written in any programming language. The language must only provide the opportunity to access Web services and needs to have native database support. Possible programming languages would be Java, Python or even scripting languages like PHP. In our proof-of-concept implementation Python is used, so the mashup can not just run on a dedicated server but also on platforms such as Google's App Engine [19].

3.1.6.1 Application Module

To ensure that only the authenticated user has access to the cluster all calls to the *job* or *filesystem* Web service are preprocessed by the application module. Information about the current authenticated user is given to the application service, so that it can ensure that the user only has access to her data.

When accessing the file system Web service the application module alters the requested path and adds the home directory of the user on the HDFS. If the user requests to list the contents of folder */data,* for example the application module changes the request to */home/<userid>/data*.

The priority scheduler uses the queue name of Hadoop MapReduce to separate the different users. In order to separate the users from each other the application module automatically adds the correct queue name to each call to the job interface of the application service. The corresponding queue name is stored in the database.

Note: Separating the users using a home folder and the queue name is not very secure. The users could design their MapReduce programs in a way that they can access the files of the whole cluster. In a production environment additional security on the Hadoop MapReduce cluster has to be installed.

3.1.6.2 Payment Module

The payment module implements an endpoint for the payment service. When the payment process is over, the payment service notifies the mashup at this endpoint and the payment module can deposit the money on the application service using the application module.

3.1.6.3 Authentication Module

The Web service of the authentication module is the endpoint for the user to authenticate herself against the mashup. The authentication module provides two different authentication methods: OpenID and signed URL.

The function *types* returns to the user all available types of authentication mechanisms that can be used for authentication purposes against the mashup. If the user chooses OpenID she gets redirected to the OpenID provider according to the process described in subsection 2.6.3. If she chooses the signed URL mechanism he uses the private key available from the user module.

3.1.6.4 User Module

The user module provides information about the currently logged in user. Information includes: billing information, the SSH public key for uploading data and MapReduce program to the cluster, and the private key for authentication perposes.

3.1.6.5 Database Module

The database module serves as a façade to the database. All persistent data, is given to the database module so that it can store the data in the database. The module provides no interface to the user but only to other modules.

3.1.7 User Interface

The user should have two different opportunities to submit his jobs: a web site using a browser and a command line script which can be used in other applications.

For accessing the service with a browser an AJAX based web site is used. All the services of the mashup can be accessed by asynchronous Web service calls. When using a browser the standard authentication method is OpenID but the user can also sign his query with his private key and thereby bypassing OpenID.

The command line interface translates all the method calls to XML and sends it to the mashup: the XML data received is passed to the standard out of the command line interface. The interface is written in Python and uses the httplib2 library [9] for the communication with the mashup. The authentication data can be passed to the command line interface in every call using the *-key* parameter or can be stored in a configuration files stored in the users home directory.

3.2 MySQL

MySQL comes in different versions while the basic version is installed on a single server, the MySQL Cluster [6] version is designed to run in a distributed environment. The cluster consists of one or more MySQL hosts and several network data bases (NDB). The NDBs are the data nodes of the cluster and the MySQL hosts serve as access point to the cluster.

3.2.1 General Setup

The cluster consists of one MySQL hosts and several NDBs where the data is stored. Depending on the load of the server additional MySQL hosts and NDBs can be added.

3.2.2 Application Service

Like Hadoop MapReduce MySQL has also no native billing or control Web service interfaces. Fortunately there are many projects available that makes it easy to add such interfaces. In a small setup the control and billing interface can run on the same machine. If the load of the cluster increases it is proposed to have the billing component on a separate host that provides no control interface. Also depending on the load more than one MySQL Node is necessary. In this case every MySQL node running a mysqld server [6] is providing the control interface, and the servers have to be load balanced [5].

3.2.2.1 Control Interface

The control interface could be added by using either PHP REST SQL [14] or DBSlayer [21]. DBSlayer allows native SQL queries and PHP REST SQL adds a RESTful Web service to the database. Unfortunately DBSlayer doesn't support multiple users in the current version so it is not usable at the moment and PHP REST SQL does not support SQL query so the usage is very limited. Thus a control interface may have to be written completely from scratch.

To implement the interface, in our proof-of-concept implementation Python is used. The Web service provides only one function: the function *query* to execute a SQL query. The function takes the query string and the SQL username as input. The Web service uses the username to access the MySQL cluster, executes the query and returns the result as an XML document. It is important that the Web service uses different usernames, otherwise no billing and no security could be ensured.

3.2.2.2 Billing Interface

To add a billing interface the UserTableMonitoring of the Google MySQL Tools [8] can be used. The SQL command SHOW USER_STATISTICS shows the used resources like traffic, storage and other, per user. Based on this data the budget component can deduct money from the budget database frequently. If the user is running out of funds it sets the MySQL user to inactive and thereby ensures that no one can use the service without paying.

The billing interface implements the functions *config*, *deposit*, *budget, info,* and *invoice*. Each function requires the queue name of the user.

Billing/config information about the billing interval (the time the spending rate gets deducted from the budget), the available nodes and the SLA of the cluster.

Billing/deposit adds or reduces the amount of available funds on the users account. It requires the amount to be added or reduced (positive values get added, negative values get reduced).

Billing/budget is used to return the available budget on the application service.

Billing/invoice requires the dates for the invoice and returns the billing information for the given time.

3.2.3 Authentication Service, Payment Service and Database

For the MySQL cloud service the same services and database are used as in the Hadoop MapReduce scenario (see 3.1.3 - 3.1.5).

3.2.4 Mashup

The mashup of the Hadoop scenario can be reused to implement the MySQL mashup. Only the application module has to be changed to match the new application service.

3.2.4.1 Application Module

The application module provides the function *query* of the application service directly to the user but only has the SQL query string as input. When forwarding the data to the control Web service of the application service, it adds the current logged-in user.

3.2.4.2 *Other modules*

Since the other services are unchanged (see 3.2.3) the implementation of the corresponding modules are the same as in the Hadoop MapReduce scenario (see 3.1.6.2-3.1.6.5).

3.2.5 *User Interface*

Major parts of the user interface of the Hadoop MapReduce service can be reused (see 3.1.7). Only the Web page for the AJAX interface has to be modified to fit the changed application module Web service interface. Since we want the user to browse the database and field where he can specify his query is offered. If the result is returned another text field gets updated with the XML result.

4. RELATED WORK

Two general kinds of related work can be distinguished for this paper. On one hand, there is related work regarding the framework itself as the papers' core contribution. This kind of related work is comparable to the framework from a provider perspective as it describes other approaches to put applications into the Cloud as commercial offerings. On the other hand, there is related work with respect to the both proof of concept implementations presented in Section 3. The discussion of this kind of related work gives insight to system properties resulting from framework usage.

The Google App Engine is the most prominent Platform-as-a-Service offering. Like our framework, it supports providers with basic application services and the means to connect them to a Cloud offering [19]. Unlike our framework, it is not dedicated to pre-existing applications in general, but the functionality to be offered is usually implemented in one of the programming languages supported by the framework (currently Java and Python). Similar constraints apply to the platform functionality of Microsoft Azure [15], which is limited to the support of .NET-programs. While in principle one could integrate pre-existing applications using a PaaS for example via Web Services, it is important to note that this is not their intended usage. At the same time, most PaaS offerings bring with them certain proprietary services for billing, authentication etc.. In this sense such platforms are not open towards third party services. Other available PaaS offerings like Skytap [20] or LongJump [13] are less restrictive with respect to the implementation of the pre-existing applications, but have other drawbacks in our setting. Skytap for example helps to build private Clouds for hosting enterprise applications. The resulting Cloud offerings are primarily specialized for inhouse use and not as commercial offerings for third parties. LongJump, in contrast to Skytap, is better suited for commercial offerings but has the above mentioned openness constraints towards third party services.

Another interesting approach to integrate pre-existing service-based systems are mashup engines [9] which allow to combine several basic services to new composite Services using simple operators. Typically, mashups are used for short-lived applications constructed by their end users. This focus on end user programming requires simplicity of the connecting operators, and the resulting restrictions disqualify mashup engines for the construction of complex commercial Cloud offerings.

Amazon MapReduce [2] and Amazon RDS [3] are Amazon's corresponding offerings to the both proof of concept implementations and thus belong to the second category of related work. Both are built on top of Amazon's IaaS computation offering EC2 and use Amazon's authentication and billing functionality. This results in the fact that they are not billed fine-grained by actual resource consumption but that they are billed by usage of EC2 instance hours with an additional fee for MapReduce management in the first case. However, the application of our framework allows for developing arbitrary billing schemes for the pre-existing applications MapReduce and MySQL respectively.

5. CONCLUSIONS

Cloud Computing allows for novel business models and sales channels for yet to develop as well as pre-existing applications. Many pre-existing applications that are capable of running in distributed environments already fulfill important prerequisites to run as cloud offerings, e.g. from scalability point-of-view. However, some of them are still missing other important features such as payment services or multi-tenancy, which are necessary to consume them in a Cloud-like fashion via public Web Services and billed by consumption.

In this paper we have presented an open framework that allows for pre-existing applications to be enhanced into commercial Cloud offerings. The current version of the framework already simplifies and accelerates the process to extend pre-existing applications with the respective required Web Service interfaces. More sophisticated services such as monitoring and logging services which are reusable generically will be added to the framework as future work and can evolve it into the nucleus of an open Platform as a Service.

REFERENCES

[1] Amazon Inc. Amazon webservices homepage. http://aws.amazon.com (Last seen: 2009-09-12), 2009.

[2] Amazon Inc. Elastic mapreduce homepage. http://aws.amazon.com/elasticmapreduce (Last seen: 2010-03-18), 2010.

[3] Amazon Inc. Relational database service homepage. http://aws.amazon.com/rds (Last seen: 2010-03-18), 2010.

[4] P. Balakrishnan, S. Thamarai Selvi, and G. Rajesh Britto. Service level agreement based grid scheduling. pages 203 –210, sept. 2008. doi: 10.1109/ICWS.2008.62.

[5] K. A. L. Coar and R. C. Bowen. *Apache cookbook*. Second edition, 2007. ISBN 0-596-52994-5 (paperback). URL http://www.oreilly.com/catalog/9780596529949.

[6] A. Davies and H. Fisk. *MySQL Clustering*. MySQL Press, 2006. ISBN 0672328550.

[7] R. J. T. Dyer. *MySQL in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc., 2005. ISBN 0596007892.

[8] Google MySQL Tools. Usertablemonitoring. http://code.google.com/p/google-mysqltools (Last seen: 2010-03-01), 2010.

[9] V. Hoyer and M. Fischer. Market overview of enterprise mashup tools. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 708–721, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89647-0. doi: http://dx.doi.org/10.1007/978-3-540-89652-4_62.

[10] D. Johnson, A. White, and A. Charland. *Enterprise AJAX: Strategies for Building High Performance Web Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007. ISBN 0132242060.

[11] I. Jrstad, T. Johansen, E. Bakken, C. Eliasson, M. Fiedler, and D. van Thanh. Releasing the potential of openid & sim. pages 1 –6, oct. 2009. doi: 10.1109/ICIN.2009.5357063.

[12] A. Lenk, T. Sandholm, M. Klems, J. Nimis, and S. Tai. Whats inside the cloud? an architectural map of the cloud landscape. In *Workshop on Software Engineering Challenges in Cloud Computing @ International Conference on Software Engineering*, 2009.

[13] LongJump. Longjump homepage. http://longjump.com (Last seen: 2010-03-18), 2010.

[14] PHPRESTSQL. Phprestsql homepage. http://phprestsql.sourceforge.net (Last seen: 2010-03-01), 2010.

[15] T. Redkar. *Windows Azure Platform*. Apress, Berkely, CA, USA, 2010. ISBN 1430224797, 9781430224792.

[16] L. Richardson and S. Ruby. *Restful web services*. O'Reilly, 2007. ISBN 9780596529260.

[17] J.-M. Sahut. Security and adoption of internet payment. pages 218 –223, aug. 2008. doi: 10.1109/SECURWARE.2008.74.

[18] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *15th Workshop on Job Scheduling Strategies for Parallel Processing @ 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[19] C. Severance. *Using Google App Engine*. O'Reilly Media, Inc., 2009. ISBN 059680069X, 9780596800697.

[20] Skytap. Skytap homepage. http://www.skytap.com (Last seen: 2010-03-18), 2010.

[21] The New York Times. Dbslyer homepage. http://code.nytimes.com/projects/dbslayer (Last seen: 2010-03-02), 2010.

[22] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009. ISBN 0596521979, 9780596521974.

[23] Yahoo Inc. Yahoo homepage. http://www.yahoo.com (Last seen: 2010-02-11), 2010.