International Conference on Computational Science, ICCS 2012

GRB_WAPI, a RESTful Framework for Grid Portals

Valerio De Luca\textsuperscript{a}, Italo Epicoco\textsuperscript{*a,b}, Daniele Lezzi\textsuperscript{c}, Giovanni Aloisio\textsuperscript{a,b}

\textsuperscript{a}University of Salento, Lecce, Italy
\textsuperscript{b}Euro-Mediterranean Center for Climate Change
\textsuperscript{c}Barcelona Supercomputing Center

Abstract

Nowadays grid portals are characterized by various and different features and are implemented in very differing programming languages and technologies, still having many structural aspects in common.

This paper describes a RESTful Web API, named GRB_WAPI, specifically developed for grid computing that encapsulates all grid control and computation logic need to build a grid portal. Through the adoption of this API a portal developer does not have to deal with grid technical details focusing just on the high level design of her system and on some other aspects that concern presentation, such as portal usability and functionality. The idea of developing a traditional library has been discarded in order to free portal developers from a particular implementation technology. Thanks to this choice the portal presentation logic can be implemented in any web technology and can be deployed on a different server. Traditional Web Services and SOAP protocol approach has been discarded in order to adopt a RESTful approach to make the Web APIs lighter and also to take advantage of some other aspects illustrated in the paper.

Keywords: Grid Problem Solving Environment; Web API; REST; Grid Portals

1. Introduction

Web services [6] have emerged as a major technology for developing automated interactions between distributed and heterogeneous applications. Various standards, such as WSDL (Web Services Description Language), UDDI and SOAP, support the definition of Web services and their advertisement to the potential user community. However “lighter-weight” approaches to service are more and more requested, especially for Web applications. Web APIs technology was born just to satisfy this necessity. In such a context mashup concept has also emerged: it indicates a way to create new Web applications by combining existing Web resources using data and Web APIs. These tools offer considerable opportunities: the availability of data as service, coupled with collaborative application deployment, can help realize the vision of the open programmable Web. Examples of Web APIs are the Google Data APIs [3], which allow programmers to create applications that read and write data from Google services.

* Corresponding author. Tel.: +39 0832 297235; fax: 39 0832 297235.
E-mail address: italo.epicoco@unisalento.it

1877-0509 © 2012 Published by Elsevier Ltd.
The motivation of the work described in this paper comes from the idea that also grid services could be delivered by an apposite set of Web APIs. Nowadays there are a lot of grid portals characterized by various and different features; they deliver a lot of grid services and are implemented in very differing programming languages and technologies. However many of them share structural aspects; for example most of them rely on the Globus Toolkit middleware and uses MyProxy as a credential repository. For these reasons a GRB Web API for building grid portals have been developed. Using them, a portal developer will not have to deal with grid technical details and will be able to manage a high level design. So a portal developer will be able to concentrate on some other aspects that concern presentation, such as portal usability and functionality.

The rest of the paper is structured as follows: section 2 compares the Web Services and REST protocols in order to motivate the adoption of the latter. Section 3 illustrates the design of the proposed API and section 4 presents a set of binding libraries made available to the developer to remotely call the GRB_WAPI. Section 5 describes a Java grid portal design as a GRB_WAPI’s use case. Section 6 concludes the paper.

2. Web Services and REST technology

A service is a software component provided through a network-accessible endpoint. A consumer invokes a service by sending a request message; the provider that receives a request sends back a response message containing some output data.

SOAP is an XML language defining message architecture and message formats, that are described in an apposite XML schema, hence providing a rudimentary processing protocol.

Web Services Description Language (WSDL) is an XML language for defining interfaces syntactically.

REST [7][8] architectural style has been introduced as a simple and lightweight alternative to SOAP Web services. It generally refers to any simple interface that uses XML (but sometimes also some other formats such as JSON or plain text) over HTTP without an additional messaging layer such as SOAP. Moreover REST often delivers better performance than SOAP and scales very well because of the web caching effect. Furthermore, thanks to its simple and agile nature, REST allows a rapid software development.

Besides their message formats, SOAP and REST have also other important differences about the protocols they can use. Nowadays SOAP messages can be exchanged using various synchronous and asynchronous transport protocols (although at the time of writing the only standardized binding uses SOAP over HTTP), while RESTful web services communicate only using HTTP, that is the standard web protocol. Another web protocol that could be adopted to build a RESTful web service is the waka [9] protocol, that has been thought as a binary, token-based replacement for HTTP and is specifically designed to match the efficiency of the REST architectural style. However it has not been taken into consideration because it is still under development.

Finally the web plays a very different role in each of the two technologies. In SOAP Web services the web is a simple transport medium for messages exchange between endpoints: the HTTP protocol is used as a tunnelling protocol to enable remote communications through firewalls; besides URIs remains the same for all operations of a service. Instead in REST approaches the web becomes the universal medium for publishing globally accessible information: applications are part of the web, because they use URIs for identifying resources, data and services.

2.1. REST architectural styles

In a REST architectural style resources are identified by URIs, which provide a global addressing space for resource and service discovery. The four HTTP methods PUT, GET, POST and DELETE make up a uniform interface for managing resources. PUT creates a new resource, which can be deleted using DELETE. GET retrieves the current state of a resource. POST transfers a new state onto a resource. REST messages are self-descriptive because resources are decoupled from their representation, which is described by some apposite metadata. Every interaction with a resource is stateless and state is explicitly specified into all HTTP requests.

There are two classes of commonly accepted best practices for developing RESTful web services. The so called Hi-REST recommendations suggests the use of all the four HTTP methods and the use of Plain Old XML (POX) for formatting the content of messages. On the other hand, Lo-REST implementations only use two of the four HTTP methods: GET for idempotent requests and POST for everything else. This decision derives by the fact that proxies and firewalls sometimes allow only HTTP connections that use these two methods. Regarding the message payload
format, Lo-REST simply enforces the use of MIME-Types, without specifying a particular data format.

The implementation of the GRB_WAPI is based on Plain Old XML, as suggested by Hi-REST, but uses only the POST HTTP method in order to avoid security issues and to overcome some limitations of the GET method, that is not suitable for requests having large amounts of input data and lets all input data appear in the text URL.

3. GRB Web API

The main goal followed in this work is focused on making portal developers free from a particular implementation technology. Thanks to this choice the portal presentation logic can be implemented in any web technology and can be on a different server. The current work starts from the mature technology named Grid Resource Broker (GRB) [1][2] developed at the University of Salento.

A REST architectural style has been adopted in the GRB Web APIs design because it is a light-weight one in comparison with Web services and SOAP protocol for developing automated interactions between distributed and heterogeneous applications. However SOAP is more mature and standardized, widely supported and transport protocol independent while REST lacks standards for security and QoS; nevertheless REST gives developers a larger freedom of choice and is easy to get started with due to the agility of the design and the lightweight approach. While SOAP follows very closely the RPC model, in REST-style architectures web requests and responses are built around the transfer of representations of resources: in fact if a client knows the resources offered by a given service, it can automatically know how to manage them. Moreover, RESTful applications maximize the use of the pre-existing, well-defined interface and other built-in capabilities provided by the chosen network protocol and minimize the addition of new application-specific features on top of it.

Thanks to this Web API set a developer should be able to build a grid portal without dealing with technical details concerning grid middleware; moreover, thanks to some binding libraries specifically developed, a programmer should not even directly take care of GRB_WAPIs invocation. So not only grid middleware but also the Web API set built on the top of it could be entirely seen as a black-box for portal development.

3.1. GRB_WAPI architecture

GRB_WAPI architecture is made up of two layer: the first is responsible for remote call input and output, the second implements the control logic that processes input data and produces the outputs.

Each GRB_WAPI, as you can see in figure 1, starts with a part that reads any input parameter and/or any uploaded input file from a HTTP request.

![Fig. 1. General internal structure of a GRB_WAPI](image)

Then a corresponding function of a grb_client library is invoked passing parameters and XML files read from HTTP request. The processing logic of this layer is based on a previously built set of GRB libraries which in turns are based on the Globus Toolkit grid middleware. The main components of GRB libraries are:

- **grb_gram** for job execution using the Globus GRAM. Non-blocking versions of the functions have been added in order to asynchronously submit, check and cancel a job;
- **grb_gridftp** for secure file and directory transfer;
- **grb_igrid** for querying the iGrid Information Service;
• `grb_mds` for querying the Globus MDS2 information service;
• `grb_lsf` for job submission with the LSF batch system;
• `grb_myproxy` for the retrieval of the user’s proxy credential.

The function returns as output an XML document written on HTTP response of the GRB_WAPI: the output element, which is the XML document root, includes a result element, which contains processing results, and an error element, which in case of error contains a message and its related code stored in id attribute.

3.2. The most relevant GRB Web APIs

This section presents some of the more important GRB_WAPIs for login, job submission and job monitoring. The first Web API that has to be invoked is `grb_wapi_login`, which authenticates a user with the grid. It takes a username and a password as input and returns in its XML response a session handle that represents the user authentication with the system. This is an ephemeral cookie (never stored on the user’s hard disk) encrypted with TLS, containing the following data: user’s login, random data, a timestamp, an expiration date, and a Message Authentication Code (MAC) for the previous data. This session id has to be passed as an input parameter to all next GRB_WAPI calls: this is necessary because, due to the stateless nature of REST technology, the connection state has to be explicitly specified in every HTTP request.

The `grb_wapi_job_submit` Web API allows to submit a job described by a JSDL file that is passed as an input in the HTTP request. If the job submission is successful, the Web API returns the job id in its XML response.

The `grb_wapi_delete_job` Web API cancels the specified job submission if it is still active. The `grb_wapi_clean_job` Web API removes unconditionally the specified job from the GRB_WAPIs’ database and removes also temporary and output files related to the specified job.

The `grb_wapi_get_status` Web API returns a list of the jobs with their status.

3.3. Performance considerations

Here some considerations are described to understand the performance improvements that can be achieved by using a REST-based design.

A typical SOAP framework, in order to serialize a SOAP message into XML, has to pass it through various handlers before finally transmitting the result to a remote server via HTTP. Instead a REST implementation allows a faster creation of a HTTP packet with a pertinent XML payload which can be immediately sent to a REST-specific URL. REST-based design has been chosen in order to incur both a lower latency and packet size than a hypothetical SOAP counterpart. Besides, as shown in [16], REST clients are generally able to resiliently process many synchronous HTTP requests while trending upwards in a strictly linear fashion, while SOAP clients shows an exponential trend.

In particular REST style is more suitable for simple CRUD operations than the SOAP implementations: most of the GRB_WAPIs deal just with such operations, which involve adding, editing and deleting of various type of items which could be applications, VO, resources, users and so on.

4. Binding libraries

In order to make GRB_WAPIs easier to use, some binding libraries have been developed, that is some sets of API which act as an abstraction layer, in some common web programming languages. In general a binding from a programming language to a library or OS service is an API providing that service in the language. Some motivations for creating library bindings are often software reuse (rather than implementing again the library in several languages) or the impossibility of implementing certain algorithms efficiently in high-level languages.

The adoption of a REST style makes easier the development of various binding libraries for different programming languages and technologies: instead of a specialized SOAP client, any HTTP client library may be used to interact with a set of REST Web APIs; REST technology also allows to take advantage of this fact, because there are only a handful of well-supported SOAP client libraries, while every modern language has a built-in HTTP library. In addition there are some help libraries such as libcurl [11], which is provided for various programming languages [12][14]. Furthermore REST binding libraries don’t need any service discovery and interpretation: they
only need the IP address or the hostname of GRB_WAPIs server they have to contact.

Java and PHP binding libraries have been developed with the aim of enabling server side technologies. A binding to Ruby has been implemented in order to take advantage of some peculiar aspects of this language, as described in the next subsection. In order to support client side developers and also to smooth the way for the adoption of mashup technology a Javascript binding library has been developed, which exploits Ajax technique to interact asynchronously with the Web API server. Thanks to the RESTful design adopted, which minimizes XML overhead, it is possible to develop thin client applications also when using a client-side Javascript implementation.

Fig. 2. General structure for binding libraries for object-oriented languages (Java, PHP, Ruby) and Javascript

4.1. Java binding library

The Java binding library follows the three-layers structure in figure 2:
- the highest layer consists of methods the developer will invoke to build his portal;
- the middle layer is a wrapper that is used by the first layer methods to invoke GRB_WAPIs;
- the lowest layer is used by the middle layer to build HTTP requests that contacts GRB_WAPIs, passing them the appropriate parameters.

`GRBWapiServer` class is the one directly used by a portal developer and is an abstraction of the Web Server where GRB_WAPIs have been installed: so the class constructor takes as an input parameter the Web Server URL, while almost all the class methods are wrappers for corresponding GRB_WAPIs invocations.

After `GRBWapiServer` object has been created, the `openConnection` method allows the connection to GRB_WAPI server and authenticates the user. If authentication has success a session handle is created: this handle will be taken as an input parameter by GRB_WAPIs invoked by all the other class methods.

As many GRB_WAPI take also some files besides simple parameters as input data, the POST method and multipart request is used in order to support file uploading in HTTP requests. As described in RFC1521 [10] in multipart request the Content-Type field headers has the multipart/form-data value and the request body is made up of more parts, delimited by an apposite boundary string that has to be specified in the Content-Type field of the main header. The `MultipartFormOutputStream` class is used by the `GRBWapi` class to write such a HTTP request.

4.2. PHP binding library

The PHP binding library leverages on the support for object oriented approach introduced in the latest versions of PHP. The binding library has the same three layer structure that characterizes the Java library described before.

After `openConnection` method of `GRBWapiServer` class has established a connection, the portal developer can store the `GRBWapiServer` object into the HTTP session, represented by the $_SESSION associative array, in order to keep track of the user’s authenticated state.
4.3. Ruby binding library

The Ruby binding library reflects the same three layer structure of Java binding library, with a significant enhancement introduced in the top layer GRBWapiServer class.

A new methods grb_wapi_index has been developed for detecting the GRB_WAPIs available on the server. By invoking grb_wapi_index, GRBWapiServer class is able to automatically discover the other GRB_WAPIs that have been installed on the specified server and it can automatically define all the methods that will wrapper the GRB Web APIs, exploiting Ruby metaprogramming features [13]. This trick allows to add or remove a GRB Web API on the server without modifying the binding library, which becomes a transparent layer of the architecture.

GRBWapiServer class constructor takes as an input parameter the Web Server URL and uses it to invoke grb_wapi_index and define its wrapper methods. The structure is analogous to the other binding libraries: the openConnection is used to authenticate the user and to initialize the session handle. This handle will be taken as an input parameter by GRB_WAPIs invoked by all the other class methods. The other methods take as an input parameter an associative array containing all the input data that have to be passed to the GRB_WAPI. These input data could be strings or XML files and are represented in the associative array as keys and values: for a string input parameter the key is the parameter name and the value refers to its value; for a file input parameter, the key corresponds to the filename and the value refers to the file content. Each of the wrapper methods creates an object that instantiates a GRBWapi class, passing to the constructor the specific Web API URL.

4.4. Javascript binding library

For the development of a Javascript binding library, both AJAX and JSON [4][5] as possible alternative techniques have been evaluated.

JSON brings some important advantages: it is lighter than XML because XML tag structure is more verbose; besides with JSON developers don’t have to deal with data parsing and they have simply to use XMLHttpRequest class for transport.

However AJAX represents a more suitable and flexible solution. AJAX data are written in XML, which is a standard format, and so they could be used by a web application as well as a desktop application without modifying server-side logic. Moreover AJAX XML stream can be also bidirectional, because not only a server can send XML to clients, but also clients can send XML to a server. It is easier to test and debug XML streams on the server rather than to test and debug a Javascript string. XML streams besides being parsed with DOM could be also transformed with XSLT: XSLT transformations can be performed server-side but also most of modern web browsers support them. For all these reasons AJAX has been chosen between these two technologies.

In the library the grb_wapi_request function uses Javascript XMLHttpRequest class to send input data to the CGI web server and receive GRB_WAPI’s XML output document. This class is the essence of AJAX technology: it allows the browser to submit both GET and POST HTTP requests to the server asynchronously and without having to entirely refresh a web page. On the top of the generic function a front-end layer has been developed, made up of specific functions, each of them representing a GRB_WAPI.

5. Case study

This section describes a grid portal developed using the Java binding library for GRB_WAPI [15]. The portal is characterized by clarity of design, efficient modularity, multiple views, ease of growth and maintenance and powerful user interfaces: Java portlets technology has been adopted to reach these goals. JSR-286 defines portlet 2.0 specification: the portal development is based on this specification, because it is compliant with Web 2.0 technologies and, in particular and because it allows handling asynchronous HTTP requests. Liferay has been chosen as portlet container, even though the portal has been designed to be container independent.

The system is based on a three-tier architecture:

- The first tier is represented by the user accessing the portal through a web browser.
- The second tier, which implements the presentation logic, consists of a portal server hosting the portlets and responsible for dynamically building web pages that containing portal contents.
- The third tier, which implements the control and computation logic, is represented by the Web Server where the
GRB WAPI logic is deployed; it includes also a MySQL server which contains a relational database accessed by GRB WAPIs.

The UML diagrams in figure 3 show the described architecture and all the interactions among its components. When a client requests a page, each portlet of that page carries out the following operations in iterative manner to get some HTML fragments which have to be assembled to make up the page content:

- It invokes a GRB WAPI remote call on the WEB API Server, which sends back an XML fragment containing elaboration result.
- It applies an XSL stylesheet to the XML fragment to execute an XSLT transformation and obtain an HTML fragment.

Fig. 3. Portal architecture

5.1. Portal framework design

The portlet development is based on the design of a framework that allows invoking methods of Java binding library and transforming XML output into portlet contents.

Fig. 4. UML class diagram for framework main section

The UML class diagram in figure 4 illustrates the main section of the portal framework. GenericPortlet abstract
class implements Portlet interface and offers a template that can be used to develop any portlet. In the proposed framework this class is further specialized into GRBPortlet abstract class, which in turn is used as a template to build the use case portals portlets. The portlets can define their VIEW and EDIT specific contents by implementing GRBViewInit and GRBEditInit methods respectively. The processGRBAction method defines the operations the portlet has to perform when a new HTTP request has been generated by a form submission. The isAjaxPortlet method is used by a portlet for asynchronous HTTP requests using Ajax technology; this allows a portlet to retrieve data from the server asynchronously in the background without interfering with the display and the behavior of the rest of the page where the portlet content lies; so, for example, a form submission in the portlet will not cause a refresh of the entire page.

In all this cases, in GRBPortlet class a template method design pattern is applied, which suggests defining the general structure of an algorithm in an abstract class allowing concrete subclasses to define some details of it. The only portlet that doesn’t extend GRBPortlet is the Login portlet, which is a direct subclass of the GenericPortlet abstract class. After the user has provided his username and password, processAction method builds a GRBSession object, which contains a GRBWapiServer instance; the openConnection method is invoked on this object to authenticate the user with the GRB WAPI server. As seen before, the content that the portlet will show in VIEW and EDIT mode can be defined as the composition of more components, which are HTML fragments generated by XSLT transformations applied to XML markups that represent GRB WAPIs outputs.

AbstractComponent abstract class represents a generic component that can be rendered in portlet VIEW or EDIT mode: it defines the toHTML abstract method, which is responsible for building an HTML fragment and writing it in the portlet content; this method has to be implemented by concrete subclasses, GRBPortletComponent, GRBPortletContent and GRBPortletMultiContent. A strategy design pattern is applied: each component is created by instantiating one of the three subclasses, but then it is considered as an AbstractComponent object; however, when toHTML method is invoked to write portlet contents, what is actually executed is an implementation specific for the subclass instantiated by the object.

GRBPortletContent subclass represents an entire content that can be rendered in portlet VIEW or EDIT mode. Content is made up of one or more components, each of them in turn represented by GRBPortletComponent class. So, in addition to the strategy design pattern, a composite design pattern is applied, which suggests to combine more objects within another object which has the same behavior of its component parts; so, this design pattern allows to handle single objects and compositions in a uniform way. In the case just described AbstractComponent class is a common abstraction for both primitive and composite objects: it is actually extended by both GRBPortletComponent class, which represents a primitive object, and GRBPortletContent class, which represents a composite object.

The GRBPortletMultiContent subclass represents multiple content, given by the composition of more simple contents, which are instances of GRBPortletContent class. Here composite design pattern is used also for the GRBPortletContent and GRBPortletMultiContent classes; in this case, GRBPortletContent represents the primitive object, while GRBPortletMultiContent represents the composite object.

All AbstractComponent subclasses use the PortletHTMLWriter class object that is passed as an input to their toHTML method, in order to write the HTML fragment of the component they represent. The GRBPortletComponent class invokes a concrete implementation of transform method of a subclass of the DocumentTransformer abstract class to execute XSLT transformations that generate HTML fragments from XML documents; this is another application of the strategy design pattern.

The GRBChartBuilder abstract class is a subclass of the DocumentTransformer class and provides an implementation of transform method that builds a chart with the data in the XML markup it takes as an input.

5.2. The Grifin project

Grid for Finance project (Griffin) has served as a realistic and complex use case for developing a grid problem-solving environment based on the GRB WAPI technology described in the previous sections. The development of efficient and effective decisional support tools in financial area is strategically important for financial institutes since they could provide great opportunities in terms of enhancement of existing applications and development of new services with a higher level of customization.

After the definition of requirements and specific constraints, through the grid portal, financial operators can obtain suggestions about portfolio composition along the investment planning horizon, according to the risk
aversion attitude of the investor. Solutions are plotted as charts depicting:
- The efficient frontier representing solutions in terms of risk measure and final wealth as function of the risk aversion attitude.
- The portfolio evolution along each scenario.

Before accessing the portal services the user has to authenticate himself through the Login portlet. User authentication is based on Shibboleth, a software tool for web single sign-on. It allows retrieving information about user identity and making informed authorization decisions for individual access of protected online resources in a privacy-preserving manner. A Shibboleth Service Provider, installed on the portal server, is responsible for redirecting the user’s request to the external URL of a Discovery Service, which in turn enables the user to select an Identity Provider. After the user provides his username and password, the Identity Provider retrieves some user data and then redirects back to the portal site. From then on the user is authenticated and the web application is able to retrieve his data from apposite request header.

![Fig. 5. (a) Job Status portlet; (b) Job Status Details portlet](image)

Griffin portlet is the portlet the user has to use to submit a job of the Griffin application. Three tabs compose the portlet user interface: Instruments, Parameters and Resume. In the Instruments tab the user can select instruments, specifying also an invested value, a lower bound and an upper bound. In the Parameters tab the user can set the initial cash, an eventual minimum return and a time period in months; he can also specify as a constraint the maximum risk for counterparts, categories and countries. The Resume tab sums up for the user the data he has inserted in the previous tabs, so he can check them before submitting the job. The GRBPortletMultiContent class previously described has been used to implement the three-tabs content of this portlet.

Through the Job Status portlet the user can monitor the jobs he has submitted. For each job the user can get the list of its subjobs (if existing), the submission timestamp, the status associated with a timestamp representing the time it entered that status. Through this portlet the user can also cancel a job execution.

The Results portlet shows the results of the Griffin application job execution for each level of the selected scenario for the selected lambda value. Result data are shown both through tables and through pie charts.

The Job Status Details portlet allows the user to check some status details about his jobs and becomes active when the user clicks on the output, the view log or the status column of the job. So, through this portlet the user can download the output files of a submitted job whose execution has ended, view the job log and, above all, he can visualize a horizontal bar chart that shows the entire status history of the job.

6. Conclusions

In this work a set of Web APIs specific for building grid portals has been described: a developer can use GRB_WAPIs as a black box which hides him grid technical details. A Web developer will be able to concentrate on other portal aspects that concern presentation, such as portal usability and functionality. REST technology is
inexpensive to acquire and thus has a very low barrier for adoption. The effort required to build a client to a RESTful service is very small as developers can test it from an ordinary web browser.

The GRB Web API approach fulfills a good trade-off between performance, modularity and reusability. A stateless RESTful web service can serve a very large number of clients, thanks to the support for caching, clustering and load balancing built into REST. Such a service can offer better performances, also due to the adoption of lightweight message formats that reduces network traffic. Besides the GRB Web API’s response messages have a modular and well defined structure, that has less overhead than SOAP messages envelope. By adopting this solution it is possible to achieve also a great software reusability and an important separation of concerns.

The GRB Web APIs represent also a partial step towards cloud technologies. Typical cloud computing providers deliver common business applications online that are accessed from another Web service or software like a Web browser, while the software and data are stored on servers. GRB_WAPI calls could be seen just as a lighter alternative to Web services. Moreover GRB_WAPIs, thanks to their independent nature (as they can be installed on a Web Server that can be different from the application server where the portal has to be deployed), could be easily turned into an on demand service.

As a further step in a cloud direction it is possible to extend and modify the GRB Web API in order to make it a suitable tool for developing more autonomous and transparent systems: as a final result hardware, software and data should be automatically handled and reconfigured within a single platform that could be presented to users. Besides enhancements towards cloud technologies should have also the aim of providing proper QoS levels for all users, in terms of CPU load and memory occupation.

However there is still another obstacle towards cloud technologies. In Service-Oriented Architectures (SOA) web services provides persistent and reliable queries for transmitting messages, allowing to interact with a service provider even when it is not available. On the contrary, since RESTful web services are characterized by synchronous interactions, they are not suitable for such failure scenarios. When a REST server is down, all HTTP requests will fail and clients will have to handle such connection timeout failures themselves.

References

4. JSON interchange format, http://www.json.org/
11. libcurl - the multiprotocol file transfer library, http://curl.haxx.se/libcurl/