REST-based SLA Management for Cloud Applications

Alessandra De Benedictis
Università di Napoli Federico II,
DIETI,
Napoli, Italy
alessandra.debenedictis@unina.it

Massimiliano Rak
Seconda Università di Napoli,
DII,
Aversa, Italy
massimiliano.rak@unina2.it

Mauro Turtur, Umberto Villano
Università del Sannio,
DING,
Benevento, Italy
{*mauro.turtur, villano*}*@unisannio.it*

***Abstract*—In cloud computing, possible risks linked to availability, performance and security can be mitigated by the adoption of Service Level Agreements (SLAs) formally agreed upon by cloud service providers and their users.**

This paper presents the design of services for the management of cloud-oriented SLAs that hinge on the use of a REST-based API. Such services can be easily integrated into existing cloud applications, platforms and infrastructures, in order to support SLA-based cloud services delivery.

After a discussion on the SLA life-cycle, an agreement protocol state diagram is introduced. It takes explicitly into account negotiation, remediation and renegotiation issues, is compliant with all the active standards, and is compatible with the WS-Agreement standard.

The requirement analysis and the design of a solution able to support the proposed SLA protocol is presented, introducing the REST API used. This API aims at being the basis for a framework to build SLA-based applications.

***Keywords*-Cloud, SLA, WS-Agreement, REST, API**

I. Introduction

In the last few years the cloud computing paradigm has widely diffused, from computer scientists to simple computers users. Apart from single users, institutions have taken great advantage from cloud characteristics such as rapid deployment of applications, flexibility and reconfiguration of employed resources, low or null start-up costs. On the other hand, it is becoming apparent that the great potential of this paradigm is partly underutilized, mostly because of doubts regarding availability, performance and security.

As the cloud paradigm is founded on the provision of everything as a service (from hardware resources to applications), the canonical way to mitigate possible risks is negotiate a set of required non-functional properties (level of Quality of Service, QoS) with the service providers. This is done through the adoption of of Service Level Agreements (SLAs), which serve as a means of documenting formally the service(s), performance expectations, responsibilities and limits between cloud service providers and their users [1], [2], explicitly taking into account obligations, service pricing and penalties in case of agreement violations. Relevant issues to be taken into account for effective SLA use are discussed in [3].

Currently the definition of SLA standards is an active field of research [4], [5], but concrete results are available only for SLA representation (e.g., WSAgreement [6], WSLA [7]), and as concerns tools/environments for their management (e.g., SLA@SOI [8], WSAG4J [9]).

The Authors of this paper are involved in a FP7-ICT programme project (SPECS[1]) which addresses cloud security through SLAs. The SPECS project intends to improve the state-of-the-art in cloud computing security by creating, promoting and exploiting a user-centric framework and a platform dedicated to offer Security-as-a-Service using a SLA-based approach, in particular with respect to negotiation, continuous monitoring and enforcement [10], [11], [12].

One of the key points of the SPECS project is the provision of a solution for the management of the whole life-cycle of SLA contracts. The design that will be presented in this paper has a sligthly wider scope, as it stands as a management solution for any type of cloud SLA, not necessarily those linked to security, which are the main object of the SPECS project. In particular, our proposal is the design of a web service for the management of cloud SLAs through a REST-based API.

The WS-Agreement Specification by the Open Grid Forum (from here onwards, WSAG) [6] defines a language to specify SLAs, and a protocol for their creation based on templates for the final agreements. We will adopt the format for SLA defined by the WSAG, but with a focus on cloud environments and security. This implies the use of an agreement protocol state diagram built according to cloud SLA standards, but enriched as compared as the original WSAG specification. Our state diagram takes explicitly into account phases like *Negotiation* and *SLA Implementation*, as well as actions like remediation and re-negotiation, which are neglected in WSAG.

Currently most implementations of WS-Agreement rely on the canonical SOA approach based on SOAP web services, the so-called WS-*. However, the use of RESTful architectures is rapidly diffusing, promising higher performance and better scalability [13]. As mentioned above, our

[1]http://www.specs-project.eu

solution is based on an open REST-based API. It is different from existing work [14], as it is focused primarily on cloud environments, and implements an expanded SLA state diagram reflecting the state-of-the-art of SLA standards.

This paper will go on as follows. In the next section, we present related work. Then the SLA life cycle is considered, presenting an extended SLA state diagram. The requirements of an API for SLA management are discussed in Section IV. The complete description of our REST API is presented in Section V. The paper closes with our conclusions and plans for future research.

II. Related work

The definition of Service Level Agreement is an active topic for standardization bodies, because they are at the interface between cloud user needs and the services and features that cloud service providers (CSPs) are able to offer. At the state of the art, there are several different proposals coming out from different institutions. The European Commission has set up a dedicated Working Group (CSIG-SLA) to address the SLA problem, in the context of the EC directive on *Unleashing the Cloud Computing* [15] and related activities. The first result obtained by this group is a guideline for standardization bodies that outlines how standards related to SLA should be organized, offering examples of the key concepts to be considered [5].

A more advanced state of SLA standardization is offered by ISO 19086 [4], which proposes a clear standard for SLAs in clouds. The above cited standard introduces the main concepts related to SLA, and examples of guarantees that can be offered. However, their description is very general and leads more to SLAs written in natural language than to automated SLA systems based on machine-readable formats.

WS-Agreement [6], born in the context of grid computing (which relies on a stable middleware, and on a well-defined way of using services and representing resources), is the only standard supporting a formal representation of SLAs and a protocol that aims at their automation. The main limit of such solution is that it was devised in a grid-oriented technological context, and that it is not completely fit in other contexts, such as clouds.

The majority of the cloud-oriented FP7 projects (Contrail[2], mOSAIC[3], Optimis[4], Paasage[5]) are inclined to adopt WS-Agreement representations, suitably adapted to the cloud context. In one case (see the paper [14] mentioned in the introduction), the implementation approach followed is similar to the one proposed in this paper, in that a REST API is developed to support the WS-Agreement protocol, even if using the “traditional” WSAG state diagram and with no particular support for clouds. The pros and cons of the adoption of a REST architecture [13] are discussed in [16] and [17]. The extension of WS-Agreement to support renegotiation is instead discussed in [18].

[2]http://www.contrail-project.eu
[3]http://www.mosaic-cloud.eu
[4]http://www.optimis-project.eu
[5]http://www.paasage.eu

However, to the best of our knowledge, none of the main commercial IaaS providers (Amazon, Rackspace, GoGRID, ...) currently offers negotiable SLAs. What they usually propose is an SLA contract that specifies simple grants on uptime percentage or network availability. Moreover, most of the providers offer additional services (for example, Amazon CloudWatch), which monitor the state of target resources (i.e., cpu utilization and bandwidth). Open Cloud Engine software like Eucalyptus, Nimbus, OpenNebula, also implement monitoring services for the private cloud provider, but do not provide solutions for SLA negotiation and enforcement. A survey of the SLAs offered by commercial cloud providers can be found in [19].

III. The SLA life-cycle

According to current standards on cloud SLAs (WS-Agreement [6], ISO19086 [4], . . .), the SLA life cycle is characterized by five phases (Figure 1), *Negotiation*, *Implementation*, *Monitoring*, *Remediation* and *Renegotiation*.

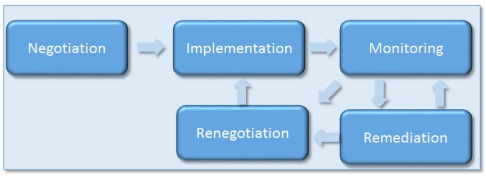

Figure 1. The SLA life cycle

During the *Negotiation* phase, a cloud service customer and a cloud service provider carry out a (possibly) iterative process aimed at finding an agreement that defines their relationship with respect to the delivery of a service. The agreement may specify both functional properties related to the identification of the service, and non-functional properties such as performance or security. Negotiation typically ends with the formal acceptance of an SLA (hereafter referred to as “signature”) by both parties, and it is followed by the implementation phase. During the *Implementation*, the CSP provisions and operates the cloud service, but also sets up and provides the customer with the processes needed for the management and monitoring of the cloud service, the report of possible failures and the claim of remedies.

After the implementation of an SLA, the *Monitoring* phase takes place. If any SLA violation occurs, i.e., if one of the agreed terms of the SLA is not respected, the cloud service customer may be entitled to a remedy (*Remediation* phase). Remedies can take different forms, such as refunds on charges, free services or other forms of compensation.

Finally, during the Monitoring and/or Remediation phases, either the cloud service customer or the cloud service

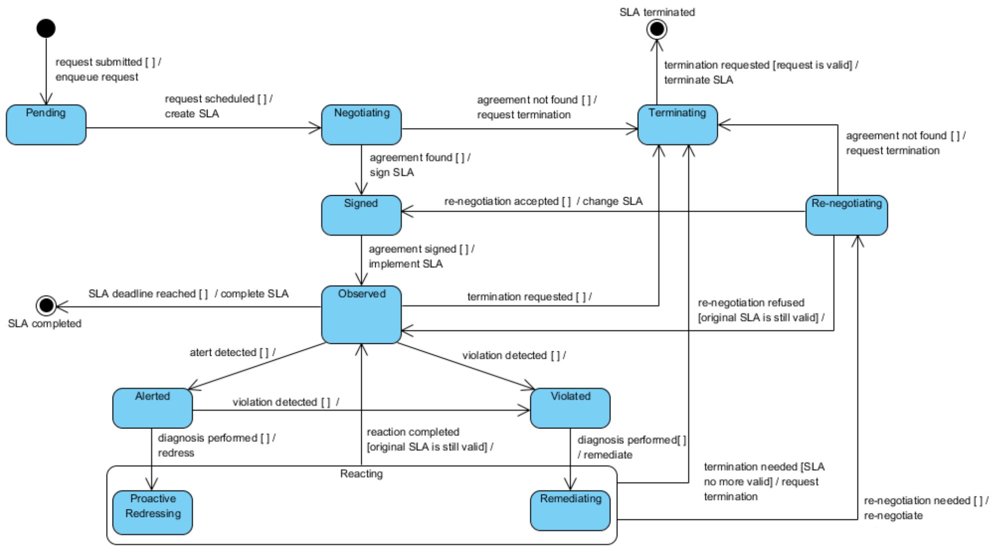
SLA terminated
request submitted [ ] / enqueue request
Pending
request scheduled [ ] / create SLA
Negotiating
agreement not found [ ] / request termination
Terminating
termination requested [request is valid] / terminate SLA
agreement not found [ ] / request termination
agreement found [ ] / sign SLA
Signed
re-negotiation accepted [ ] / change SLA
Re-negotiating
agreement signed [ ] / implement SLA
SLA deadline reached [ ] / complete SLA
SLA completed
Observed
termination requested [ ] /
re-negotiation refused [original SLA is still valid] /
atert detected [ ] /
violation detected [ ] /
Alerted
violation detected [ ] /
Violated
diagnosis performed [ ] / redress
reaction completed [original SLA is still valid] /
diagnosis performed[ ] / remediate
Reacting
termination needed [SLA no more valid] / request termination
Proactive Redressing
Remediating
re-negotiation needed [ ] / re-negotiate

Figure 2. Refined SLA life cycle

provider may require a change in the SLA (e.g., if a service provider permits variable terms). This may lead to a *Re-negotiation* phase, changing the original SLA terms.

In Figure 2, we show an SLA state diagram that takes into account the above presented phases and enriches them. In particular, it introduces explicitly the concepts of SLA alert and proactive reaction to prevent violations, and so enables a full audit over the SLA evolution for the benefit of both service customers and providers.

The UML state machine in Figure 2 outlines aspects related to possible violations, pro-active reactions and re-negotiations of the SLAs that are supported by state-of-the-art standards. It should be pointed out that the proposed diagram is compliant with the one proposed in the WS-Agreement specification, while introducing some improvements. Compared to our diagram, the WSAG "original" diagram (Figure 3) hides the scheduling of the negotiation process and the negotiation itself in the *pending* state (i.e., an SLA is pending during all the negotiation process), while we explicitly differentiate between these two states by introducing the *negotiating* state. Moreover, the *pending* state in the WSAG diagram completely hides the Implementation phase, while we make an explicit reference to the achievement of the agreement, which is ratified through the signing of the SLA and triggers the SLA implementation (*signed* state). Once signed, and after that the Implementation has taken place, the SLA enters the *observed* state, corresponding to the Monitoring phase.

In the diagram proposed in Figure 2, we separately represent the management of SLA alerts and violations. During the Monitoring phase, the cloud service provider may check for deviations from the desired behavior that do not directly cause the violation of an SLA, but that may likely induce it in the near future, if not properly handled. In such situations, the SLA is set in the *alerted* state, where a diagnosis activity is performed to determine the root cause of the alert and proper countermeasures are identified (e.g., a reconfiguration of the service being delivered), which are later on enforced while the SLA is in the *proactive redressing* state.

Once the countermeasures have been applied and the alert is no longer active in the system, the SLA returns to the *observed* state. If instead a violation occurs, and is detected while in the *observed* state or even in the *alerted* state, suitable remedies are applied in the *remediating* state, as discussed earlier in this section.

During the *Monitoring* and/or *Remediation* phases, either the customer or the provider may require a change in the SLA. In order to represent this situation, we devised a transition from the *reaction* macro-state (including both the *proactive redressing* and *remediating* states) and from the observed state, towards the *re-negotiating* state.

Finally, an SLA may enter the *terminating* state if (i) an explicit termination request has been issued by either parties, (ii) an agreement has not been found in the negotiation or re-negotiation phases, (iii) a detected SLA violation implies the termination of the SLA.

IV. Requirement analysis of the API for SLA management

In this section we discuss briefly the functional requirements that must be satisfied to enable the management of the SLA life-cycle illustrated in Section III. The solution we aim at designing must allow for the management of SLAs from

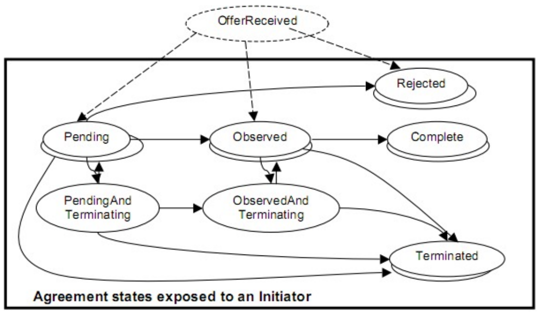

Figure 3. WS-Agreement State Diagram [6]

their negotiation (or re-negotiation) to their termination, by supporting their implementation and by handling possible alerts/violations that may arise during their monitoring.

The main design requirement is that all information related to SLAs and their state has to be stored in an SLA repository, and that specific services should be available to query and to update the contents of such repository. In the presence of an alert or violation, the API should allow to update the state of the involved SLAs.

The API to be designed should allow to create SLAs (possibly by means of a template), to store them in an SLA repository, to retrieve them based on their ID, and to get other relevant information such as the list of possible states that an SLA can traverse. It must allow the change of an SLA that is in the *Negotiation* or *Re-negotiation* phase, and the annotation of an SLA with meta-data and additional information useful for auditing purposes.

The API has also to support the transition between two states of the diagram (update of SLA state), and to handle specific activities such as the signing of an SLA (which includes a state update) and the management of events related to alerts and violations. As discussed previously, such events imply entering either the *alerted* or the *violated* states, but also include the acquisition from the system of additional information for the correct handling of the event.

V. A REST API for SLA Management

In this section we present the complete set of API calls designed to meet the requirements discussed in Section IV for the management of the SLA life-cycle. Such calls can be easily integrated into existing cloud applications in order to drive service delivery on the basis of the SLAs agreed with service customers.

In his "Maturity Model" [20], Richardson classifies the APIs for services on the web in three incremental maturity levels, according to the support offered for URIs (L1), for HTTP methods (L2), and for hypermedia (L3). Below these levels (i.e., at L0), none of the them is supported. According to the Maturity Model, RESTful APIs [21] require level L3 as a prerequisite [22]. At the state of the art, only few APIs respect such requirement; the majority of the commonly used ones is at L2. Our API sits at L2, since our design addresses both the use of URIs (to identify the resources) and the full semantic of HTTP methods (to operate on them). We plan to introduce the hypermedia support in the next API release.

In the next subsections, we first present both the REST *Resources* our API accesses and manages, and then the operations that it allows to perform on such resources.

A. SLA API: REST Resources

The SLA API we propose manages the following types of resources:

- **SLA**: it identifies a *WSAG Offer*-compliant XML document. Each SLA includes the following information: (i) the current state of the agreement; (ii) the list of occurred violations; (iii) the list of occurred alerts, and (iv) the list of associated annotations;
- **SLA template**: it identifies a *WSAG Template*-compliant XML document;
- **SLA state**: it identifies one of the possible SLA states (see Section III);
- **Alert**: it identifies an *alert*;
- **Violation**: it identifies a *violation*;
- **Annotation**: it identifies an *annotation*.

Every resource may be represented both in XML (`application/xml` mediatype) and in JSON (`application/json` mediatype) format; the client can negotiate the preferred one via HTTP headers. The only exception is represented by the *SLA* resource, which is available only in the XML format. We manage collections of resources by means of a generic envelope for XML and an array of elements for JSON. Inside the envelope/array, each item of the collection is represented by its URL.

B. SLA API: REST URLs

A call to the REST API is identified by an URL and an HTTP method, used to realize one of the CRUD operations (Create, Read Update, Delete) on resources. The call URLs use the `cloud-sla` base path, followed by the name of the resource they operate on. Parts of URLs surrounded by curly braces identify a variable string (usually an identifier). If not otherwise specified, the client can request and send resources both in XML and JSON format.

The main resources accessed and managed by our API are SLAs, which can be reached through the `cloud-sla/slas` URL. The following calls allow to retrieve available SLAs, and to add new elements to the collection of SLAs:

- **/cloud-sla/slas**
  - **GET**: it returns the available collection of SLAs. The result can be restricted (to perform a search operation) by using a suitable query string.

  - **POST**: this call adds a new SLA Offer to the collection of SLAs, and returns the URL of the created resource. The POST entity body has to contain a valid SLA Offer in XML format. The new SLA is created in the *pending* state. If no errors occur the offer is scheduled for negotiation, and the state evolves automatically to *negotiating*.
- **/cloud-sla/slas/{sla-id}**
  - **GET**: it retrieves the SLA identified by the `sla-id` variable. The SLA can be retrieved only using the `application/xml` mediatype.
  - **PUT**: it modifies the SLA content identified by the `sla-id` variable. This call is allowed only while in the *negotiating* or *re-negotiating* states.

Note that we do not provide a method to explicitly delete an item from a collection since, for auditing purposes, we assume that all the SLAs created are maintained in the future, even in case of failed negotiation or termination.

In the following, we list the calls that have been designed to perform specific actions on an SLA (identified by the `sla-id` variable).

- **/cloud-sla/slas/{sla-id}/state**
  - **GET**: it retrieves the current state of the SLA.
  - **PUT**: it updates the state of the SLA according to the value specified in the request entity body. Possible values are the following:
    - **sign**: the SLA's state evolves from *negotiating* or *re-negotiating* to *signed*.
    - **observe**: the SLA's state evolves from *signed* to *observed*.
    - **renegotiate**: the SLA's state evolves from *proactive redressing* or *remediating* to *re-negotiating*.
    - **complete**: the SLA's state evolves from *observed* to *SLA completed*.
    - **terminate**: a termination request is issued and the SLA's state is set to *terminating*. If no errors occur, the state evolves automatically to *SLA terminated*. The method invocation is allowed only in the *negotiating*, *re-negotiating*, *proactive redressing*, *remediating* or *observed* state.
    - **remediating**: the SLA's state evolves from *violated* to *remediating*.
    - **redressing**: the SLA's state evolves from *alerted* to *proactive redressing*.

By accessing the base URL for SLA resources, it is possible to reach additional resources related to the SLA, such as alerts, violations and annotations (i.e., labels and notes that customers and providers can associate to SLAs for various purposes), by invoking the following calls:

- **/cloud-sla/slas/{sla-id}/alerts**
  - **GET**: it retrieves the collection of the `Alerts` associated to the SLA.
  - **POST**: it allows to signal that an `Alert` occurred. The method, if called in the *observed* state, sets the SLA's state to *alerted*, creates an `Alert` resource associated to the SLA and returns the URL of the created resource. Otherwise, the `Alert` is embedded and stored in a new `Annotation` resource for auditing purposes.
- **/cloud-sla/slas/{sla-id}/alerts/{alert-id}**
  - **GET**: it retrieves the SLA `Alert` identified by the `alert-id` variable for the SLA.
- **/cloud-sla/slas/{sla-id}/violations**
  - **GET**: it retrieves the collection of the `Violations` associated to the SLA.
  - **POST**: it allows to signal that a `Violation` occurred. The method, if called in the *observed* or in the *alerted* state, sets the SLA's state to *violated*, creates a `Violation` resource associated to the SLA and returns the URL of the created resource. Otherwise, the `Violation` is embedded and stored in a new `Annotation` resource for auditing purposes. The POST entity body has to contain a valid `Violation`, both in XML or JSON format.
- **/cloud-sla/slas/{sla-id}/violations/{violation-id}**
  - **GET**: it retrieves the SLA `Violation` identified by the `violation-id` variable for the SLA.
- **/cloud-sla/slas/{sla-id}/annotations**
  - **GET**: it retrieves the collection of the `Annotations` for the SLA.
  - **POST**: it allows to create a new `Annotation`.
- **/cloud-sla/slas/{sla-id}/annotations/{annotation-id}**
  - **GET**: it retrieves the SLA `Annotation` identified by the `annotation-id` variable for the SLA.

Finally, in order to support the template-based SLA negotiation, as the one supported in WS-Agreement, the API includes the following calls that make it possible to manage SLA template resources:

- **/cloud-sla/templates**
  - **GET**: it retrieves the collection of defined SLA Templates.
  - **POST**: it allows to create a new `Template` and returns the URL of the created resource.
- **/cloud-sla/templates/{template-id}**
  - **GET**: it retrieves the SLA `Template` identified by the `template-id` variable.

Concurrency issues are addressed using the appropriate HTTP headers: the server responds to GET requests with the "Last-Modified" header, while the client has to perform PUT operations adding the "If-Modified-Since" header.

VI. Conclusions and Future Work

In this paper we have proposed a solution for SLA management in clouds that relies upon a suitably defined SLA life cycle supported by a REST API. The proposed SLA life-cycle is based on current standards on cloud SLAs and is compliant with the WS-Agreement specification, but introduces some important enhancements. These are related to the management of SLA alert and proactive reaction to prevent violations, and enable a full audit on SLA evolution for the benefit of both service customers and providers.

The implementation we offer for the proposed REST API (*SLA manager*), able to support the proposed SLA life-cycle, is available on line[6] and can be easily integrated into existing applications to enable the construction of software solutions for orchestrating services based on SLAs.

The solution presented can be a basic buiding block of a framework for SLA-based service management, being compliant with current and evolving standardization outcomes in this field. In the near future, we plan to integrate our implementation of the proposed REST API with a simple broker, in order to build up an SLA-based brokering service. Moreover, we are also about to integrate our SLA manager with monitoring and enforcement solutions able to enrich the offered services according to the user requirements, expressed at the time of SLA negotiation.

Acknowledgment

This research is partially supported by the grant FP7-ICT-2013-11-610795 (SPECS)

References

[1] Cloud Standard Customer Council - CSCC, "The CSCC practical guide to cloud service level agreements," April 2012.

[2] T. Trappler, "If it's in the cloud, get it on paper: Cloud computing contract issues," http://www.educause.edu/ero/article/if-its-cloud-get-it-paper-cloud-computing-contract-issues, June 2010.

[3] A. L. Diaz, "Service level agreements in the cloud: Who cares?" http://www.wired.com/2011/12/service-level-agreements-in-the-cloud-who-cares/, december 2011.

[4] International Organization for Standardization, "ISO/IEC NP 19086-1. Information Technology–Cloud computing–Service level agreement (SLA) framework and technology–Part 1: Overview and concepts," 2014.

[5] European Commission – C-SIG (Cloud Select Industry Group) subgroup, "Cloud Service Level Agreement Standardisation Guidelines," June 26 2014.

[6] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, "Web services agreement specification (WS-Agreement)," in *Global Grid Forum*. The Global Grid Forum (GGF), 2004.

[6]https://bitbucket.org/specs-team/specs-sla_platform-sla_manager-sla-api

[7] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, no. 1, pp. 57–81, 2003.

[8] M. Comuzzi, C. Kotsokalis, C. Rathfelder, W. Theilmann, U. Winkler, and G. Zacco, "A framework for multi-level SLA management," vol. 6275, pp. 187–196, 2010.

[9] "Wsag4j project web site." [Online]. Available: http://wsag4j.sourceforge.net

[10] M. Rak, N. Suri, J. Luna, D. Petcu, V. Casola, and U. Villano, "Security as a service using an SLA-based approach via SPECS," in *Proc. of CloudCom, 2013 IEEE 5th Int. Conf. on*, vol. 2, Dec 2013, pp. 1–6.

[11] V. Casola, A. De Benedictis, M. Rak, and U. Villano, "Preliminary design of a platform-as-a-service to provide security in cloud," in *CLOSER 2014 - Proc. of the 4th Int. Conf. on Cloud Computing and Services Science, Barcelona, Spain, April 3-5, 2014.*, 2014, pp. 752–757.

[12] M. Rak, J. Luna, D. Petcu, V. Casola, and U. Villano, "Security as a service using an SLA-based approach via SPECS," in *Proc. of Cloudcom*, 2013, pp. 100–105.

[13] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[14] R. Kübert, G. Katsaros, and T. Wang, "A RESTful implementation of the WS-Agreement specification," in *Proceedings of the Second International Workshop on RESTful Design*, ser. WS-REST '11. New York, NY, USA: ACM, 2011, pp. 67–72.

[15] "Unleashing the potential of cloud computing in europe - what is it and what does it mean for me?" [Online]. Available: http://europa.eu/rapid/press-release_MEMO-12-713_en.htm

[16] S. Weerawarana, "WS-* vs. REST: Mashing up the truth from facts, myths and lies," *QCon San Francisco*, 2007.

[17] C. Pautasso, "REST vs. WS-* comparison." [Online]. Available: http://goo.gl/qnRmNZ

[18] S. Sharaf and K. Djemame, "Enabling service-level agreement renegotiation through extending WS-Agreement specification," *Service Oriented Computing and Applications*, pp. 1–15, 2014.

[19] L. Wu and R. Buyya, *Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*. IGI Global, USA, 2011, ch. Service Level Agreement (SLA) in Utility Computing Systems.

[20] J. Webber, S. Parastatidis, and I. Robinson, *REST in Practice, Hypermedia and Systems Architecture*. O'REILLY, 2010.

[21] R. T. Fielding, "REST APIs must be hypertext-driven," 2008. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[22] M. Fowler, "Richardson maturity model: steps toward the glory of REST," 2010. [Online]. Available: http://martinfowler.com/articles/richardsonMaturityModel.html