This is the pre-peer reviewed version of the following article: SLAs for cloud applications: Agreement protocol and REST-based implementation by A. De Benedictis, M. Rak, U. Villano, which has been published in final form in *International Journal of Grid and Utility Computing* and is available at http://www.inderscience.com/filter.php?aid=85910 DOI: 10.1504/IJGUC.2017.085910
SLAs for Cloud Applications: Agreement Protocol and REST-based Implementation

Abstract—Users with critical data are still reluctant to move apps and data to commercial clouds, showing a substantial lack of trust in providers. Possible risks linked to availability, performance and security can be mitigated by the adoption of Service Level Agreements (SLAs) established among cloud service providers and their customers. User communities are pressing for the adoption of SLAs in cloud environments, and standardization bodies are working to identify a set of concepts that could be used to compare alternative offers.

This paper presents the design of services for the management of cloud-oriented SLAs by means 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 on security, 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 has been adopted for the implementation of an SLA Manager that can be easily integrated in cloud applications. However, it can be also adopted in different environments, as it is flexible with respect to the SLA machine-readable representation and compliant with all existing standards.

Keywords—Cloud, SLA, WS-Agreement, REST, API, SLA management

I. INTRODUCTION

In the last decade, cloud computing has become a mainstream solution for the provision of infrastructure, platform and software services. The possibility to resort to a reserve of apparently infinite low-cost compute resources in an on-demand fashion has attracted individual users, industries and institutions, letting them free to focus their efforts on software development. However, users with critical data (e.g., financial, medical and sensitive data) are still reluctant to move apps and data to commercial clouds, showing a substantial lack of trust in providers, as regards availability, performance and security issues.

As the cloud paradigm is founded on the provision of every resource (from hardware appliances to applications) as-a-service, the capability of negotiating even non-functional properties (level of Quality of Service, QoS) with service providers, to be activated on-demand as any other service, is desirable. This is enabled by the adoption of Service Level Agreements (SLAs), which serve as a means of formally documenting delivered services and related performance (and QoS) expectations, in addition to explicitly taking into account responsibilities, obligations, service pricing and penalties in case of agreement violations [1], [2]. As discussed in [3], in order to be effective, cloud SLAs should include parameters such as availability, security/privacy, disaster recovery expectations, data location, data access and portability, and several other concepts related to the management of possible issues arising during service provisioning.

Even if it is clear that the adoption of SLAs would bring benefits to both involved parties (i.e., the service customer and provider), their use by existing cloud providers is still limited. As a matter of fact, the definition of SLA standards is an open field of research [4], [5], but concrete results are available only for SLA representation (e.g., WSAGreement [6], WSLA [7]), and for what concerns the development of 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 (SPECS1), 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].

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, along with its prototype implementation, has a slightly 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 and implementation 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 agreement. As a first contribution, we extended the WSAG Specification by enriching the proposed state diagram for the representation of the SLA life-cycle. As thoroughly illustrated in the following, our state diagram takes explicitly into account phases as Negotiation and SLA Implementation, as well as actions as remediation and re-

1http://www.specs-project.eu
negotiation, which are neglected in WSAG.

Moreover, as mentioned before, we proposed a REST-based API for the management of SLAs according to this more detailed life-cycle. Currently, most implementations of WSAG rely on the canonical SOA approach based on SOAP web services, the so-called WS-. Examples of these implementations can be found in several EU projects, as for example IRMOS\(^2\), BREIN [12], Contrail\(^3\), and mOSAIC\(^4\). However, the use of RESTful architectures is rapidly diffusing, promising higher performance and better scalability [13], and a RESTful implementation of the WSAG Specification has already presented in [14]. Nevertheless, our proposal is substantially different from existing work, 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. An example of use of the API is shown in Section VI. The prototype implementation of the API, developed in the context of the the SPECS project, is discussed in Section VII. 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 standard for the adoption of SLAs in clouds. The above cited standard introduces the main concepts related to SLAs, 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\(^5\), mOSAIC\(^6\), Optimis\(^7\), Paasage\(^8\)) 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].

In service-oriented and cloud environments, several proposals addressing the negotiation and monitoring of SLAs have appeared [19], [20]. A survey focusing on the renegotiation of SLA in clouds is provided in [21]. The problem of the provision of quality of service and SLA facilities on top of unreliable, intermittent cloud providers is discussed in [22]. Recent proposals in the context of the mOSAIC project tackle the problem of offering user-oriented SLA services to cloud users [23], [24].

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 and 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 [25] and [26].

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.

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, in which the cloud services covered by the SLA are monitored in order to enable both the cloud service provider and the cloud service customer to verify the respect of the SLA constraints. 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. Other forms of remedies may require that the customer is timely notified and that the cloud service provider reacts to the service failure within a certain time frame. Finally, a remedy may be represented by any effort to alleviate the harm caused by the failure and may result in applying corrections to the system in order to avoid future violations. Finally, during the Monitoring and/or Remediation phases, either the cloud service customer or the cloud service 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 this case, up to the signature of the new SLA, the original one is still valid, but once the new SLA is signed, it must be implemented in the place of the previous one.

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. Such state is reached when the actual negotiation process is started, and implies the creation of an SLA object, to which an unique ID is
assigned. 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; alerts represent risky situations that may lead to the violation of some terms included in the SLA. Indeed, 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 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.

An overview of the requirements that drove the design of our REST API is illustrated in the UC diagram in Figure 4, where the Actor represents a generic API user.

As shown in the figure, the API should provide functionalities for:

- **Creating new SLAs.** An SLA must be created and associated with a customer when the negotiation process starts. The SLA may be built based on a template, provided by the CSP and including the set of negotiable features for each available service. Therefore, if needed, the API should provide support for the management of templates.
- **Deleting SLAs.** If errors occur during the negotiation process, an SLA that has not yet been signed should be deleted.
- **Updating existing SLAs.** When an SLA is in the negotiating or re-negotiating state, it must be possible to update its content to reflect ongoing changes. At the end of the negotiation process, the SLA must be updated with the content of the SLA Offer selected by the customer.
• Accessing and retrieving stored SLAs. Accessing and retrieving stored SLAs must be allowed during all phases of the SLA life-cycle.
• Accessing and updating the state of a stored SLA. It must be possible to update the state of an SLA during the different phases according to the state diagram reported in Section III.
• Adding and updating annotations to a stored SLA. In order to enable an easy processing of SLAs, it must be possible to annotate them with additional information.
• Storing and retrieving information on alerts/violations associated to an SLA. The API may offer functionalities for archiving and retrieving information about occurred alerts and violations for further processing, if no specific component is designed for this purpose.

V. A REST API for SLA MANAGEMENT

In this section, we present the complete API designed to meet the requirements discussed in Section IV for the management of the SLA life-cycle, named SLA API. It 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” [27], Richardson classifies the APIs for services on the web (i.e., for software services built on top of the HTTP protocol) 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 [28] require level L3 as a prerequisite [29]. 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 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 manages the following types of resources:
• SLA: it identifies a WSAG Offer-compliant XML document, provided with a unique ID and referring to a specific End-user.
• SLA template: it identifies a WSAG Template-compliant XML document used as a guideline for negotiation. Templates are built by the service provider, refer to specific types of services (e.g., web server, storage, etc.) and include all parameters that can be negotiated related to such services.

• SLA state: it identifies one of the possible states of an SLA (according to the life-cycle illustrated in Section III).
• Alert: it identifies an alert occurred in the system related to an SLA.
• Violation: it identifies a violation occurred in the system related to an SLA.
• Annotation: it identifies an annotation occurred in the system related to an SLA.

Our implementation supports different mediatypes for the representation of resources. In particular, we provide both the XML (application/xml mediatype) and the JSON (application/json mediatype) format for all resources, except for the SLA resource that is naturally represented in XML. The client can negotiate the preferred format via HTTP headers. Collections of resources are managed 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 URI. We defined suitable data models for the representation of resources in the supported formats as part of the API documentation. In Listing 1 we report, as an example, the data model proposed for the representation of the SLA state resource, available at [30]. The reader is referred to the WSAG specification for the data model of the SLA resource.

Listing 1. SLA state data model

```xml
<?xml version="1.0" encoding="UTF-8"?>
<x:schema xmlns:x="http://www.w3.org/2001/XMLSchema" >
  <xs:element name="SLAstate" type="SLAstateType"/>
  <xs:complexType name="SLAstateType">
    <xs:sequence>
      <xs:element name="sla-id" type="xs:string" minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
  <xs:schema xmlns:x="http://www.w3.org/2001/XMLSchema" >
    <xs:restriction base="xs:string">
      <xs:enumeration value="negotiating"/>
      <xs:enumeration value="signed"/>
      <xs:enumeration value="re-negotiating"/>
      <xs:enumeration value="terminating"/>
      <xs:enumeration value="terminated"/>
      <xs:enumeration value="alerted"/>
      <xs:enumeration value="violated"/>
      <xs:enumeration value="remediating"/>
      <xs:enumeration value="proactive-redressing"/>
    </xs:restriction>
  </xs:schema>
</xs:schema>
```

B. SLA API: REST URIs and calls

According to the REST specification, each REST resource is identified by a URI composed of a base-path
and a resource identifier. A resource identifier may contain variable parameters, reported in the documentation as strings surrounded by curly braces. REST API calls are specified through related HTTP methods (PUT, GET, POST, DELETE), which perform the traditional CRUD (Create, Read, Update, Delete) operations on resources.

The SLA API uses the cloud-sla base path and manipulates the resources described in Section V-A. In the following, we provide an overview of the supported API calls. The interested reader is referred to the complete API documentation, available at [31], for syntax issues and for a detailed description of data models and response code semantics.

The main resources accessed and managed by our API are SLAs. 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 that specifies the number or interval of elements to retrieve.
  - POST: this call adds a new SLA Offer to the collection of SLAs, and returns the URI 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 to negotiating.

In the following, we list the calls that have been designed to manage specific SLAs (identified by the sla-id variable), their current state and related annotations.

- `/cloud-sla/slas/{sla-id}`
  - GET: it retrieves the SLA identified by the sla-id variable. The SLA can be retrieved only if it has not been signed yet.
  - 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.
  - DELETE: this call deletes the SLA identified by the sla-id parameter if it has not been signed yet.

- `/cloud-sla/slas/{sla-id}/state`
  - GET: it retrieves the current state of the SLA identified by the sla-id variable.
  - PUT: it updates the state of the SLA identified by the sla-id variable according to the transition specified in the request body (initial and final state are specified in a proper format). It should be noted that the only allowed transitions are those defined by the SLA state diagram of Figure 2.

- `/cloud-sla/slas/{sla-id}/annotations`
  - GET: it retrieves the collection of the Annotations for the SLA identified by the sla-id variable.
  - POST: it allows to create a new Annotation. The POST entity body has to contain a valid Annotation, represented either in XML or JSON format.

- `/cloud-sla/slas/{sla-id}/annotations/{annotation-id}`
  - GET: it retrieves the SLA Annotation identified by the annotation-id parameter for the SLA identified by the sla-id identifier.

As mentioned in Section IV, for completeness sake our SLA API includes also functionalities for managing the information related to SLA alerts and violations. In fact, this information is meant to be processed by specific dedicated components for diagnosis and reaction purposes, and its management may be performed by custom components as well. Nevertheless, in the following, we illustrate the basic API calls for storing and retrieving alerts and violations associated with an SLA.

- `/cloud-sla/slas/{sla-id}/alerts`
  - GET: it retrieves the collection of the alerts associated to the SLA identified by the sla-id variable.
  - 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 URI of the created resource. The POST entity body has to contain a valid alert, represented either in XML or in JSON format.

- `/cloud-sla/slas/{sla-id}/alerts/{alert-id}`
  - GET: it retrieves the SLA alert identified by the alert-id parameter for the SLA identified by the sla-id identifier.

- `/cloud-sla/slas/{sla-id}/violations`
  - GET: it retrieves the collection of the violations associated to the SLA identified by the sla-id variable.
  - POST: it allows to signal that a violation occurred. The method, if called in the observed or in the violated state, sets the SLA’s state to violated, creates a violation resource associated to the SLA and returns the URI of the created resource. The POST entity body has to contain a valid violation, represented either in XML or in JSON format.

- `/cloud-sla/slas/{sla-id}/violations/{violation-id}`
  - GET: it retrieves the SLA violation identified by the violation-id parameter for the SLA identified by the sla-id identifier.

Finally, in order to support the template-based SLA negotiation in compliance with the WS-Agreement specification,
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. The POST entity body has to contain a valid template in XML format.

- **/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. AN EXAMPLE OF SLA REST API USAGE

In this section, we illustrate the usage of the SLA REST API for the management of the complete life-cycle of an SLA related to a generic cloud service. In order to discuss a concrete architecture, we refer to the SPECS framework, whose architecture is sketched in Figure 5 and presented in detail in [10], [11]. The SPECS framework is composed of three Core modules, devoted respectively to Negotiation, Enforcement and Monitoring. Core services run on top of the SPECS Platform, which includes the SLA Platform, responsible for the management of the SLA life-cycle, and the Enabling Platform, in charge of the deployment and execution of needed components on cloud resources. Core services inter-operate through the SLA Platform services and are orchestrated by a SPECS application. The application helps the SPECS customers negotiate desired services and acts as a gateway between customers and external service providers, whose offered services are enhanced with guarantees stated in formal SLAs.

The API presented in this paper is offered by the SLA Manager component of the SLA Platform and is invoked by the Core modules during the main SLA life-cycle phases. Figure 6 shows the sequence of invocations during the negotiation phase. It should be noted that we hide the existence of a Negotiation module in the SPECS Application and Core Modules UML life line, as an extensive discussion of the framework behavior is out of the scope of this paper. As shown in the figure, when the End-user starts the negotiation, the available templates are retrieved and are used for building possible service offers to display to the End-user. The End-user selects the service (i.e., the template) he/she is interested in (e.g., the template built for a web server service), and a new SLA is created in the pending state based on the selected template. If no errors occur, when the End-user submits his/her requirements the SLA state is updated to negotiating, and an SLA Offer is prepared based on what can be actually offered with respect to End-user requests. This negotiation process is iterative, and hopefully terminates with the formal acceptance of an SLA offer by the End-user, resulting in a further change of the SLA state to signed. At this point, the SLA is implemented.

![Figure 5. SPECS high-level architecture](image)

![Figure 6. Negotiation phase](image)

Figure 7 illustrates the sequence of invocations in the phases of enforcement and monitoring. As shown, the signed SLA is implemented and suitable monitoring systems are configured and activated, letting the SLA state be updated to observed. If an abnormal condition is detected by the monitoring system, a monitoring event is generated and processed by the SPECS framework. In case a violation is detected, the state of the SLA is updated to violated and proper remedies are applied (e.g., computation and application of penalties). If it is not possible to recover from the violation, the SLA is terminated and the End-user is notified.
VII. SLA API IMPLEMENTATION

As mentioned in the previous section, the SLA API is offered by the SLA Manager, developed in the framework of the SPECS project. As the code produced is publicly available and lends itself to be used in different contexts, the SPECS implementation of the API will be briefly described in the following.

The SPECS implementation of the SLA Manager is represented by a web application made of two components: the first one represents the back-end that handles the persistence of all data, while the second one is the front-end that exposes a REST API. Figure 8 shows the software organization of the component, including (i) the SLA Manager API, which offers the SLA APIs described in Section V, and (ii) the SLA Manager component, implementing the SLAManager interface used by the SLA Manager API component.

The SLA Manager web application is described through two different class diagrams. The first diagram (Figure 9) represents all the packages and classes that compose the SLA Manager component, while the second one (Figure 10) represents the packages and the classes of the SLA Manager API component.

The development of the web services exposed by the REST API interface has been performed using the Jersey RESTful Web Services framework. All the REST resources are mapped under the path “/cloud-sla/*”. This is explicated in the web.xml file.

Moreover, all the REST resources are represented by specific Java classes under the package “eu.specsproject.slaplatform.slamanager.restfrontend”. This is also explicated in the web.xml file.

The Figures 9 and 10 show an example of use of the calls used to retrieve an SLA collection, and to obtain a particular SLA from it.

The implementation of the SLA API is publicly available at [31].9

As previously mentioned, this work is part of the activities carried out in the SPECS project, and the API has been developed by the SPECS Consortium.

9As previously mentioned, this work is part of the activities carried out in the SPECS project, and the API has been developed by the SPECS Consortium.
VIII. 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 publicly available 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 building 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.

REFERENCES


