SLA Template Filtering: A Faceted Approach

Katerina Stamou, Verena Kantere and Jean-Henry Morin
Institute of Services Science, University of Geneva - HEC, Switzerland
Email: aikaterini.stamou, verena.kantere, jean-henry.morin @unige.ch

***Abstract*—With the commoditization of cloud computing, more and more companies prefer to outsource IT resources into virtual infrastructures. Service Level Agreements (SLAs) can be helpful to make the right investment decision. A SLA template represents the pre-agreed SLA state. A service provider proposes the SLA content and submits the template to a marketplace for customer consideration. Customers use SLA template views as "What You See Is What You Get" (WYSIWIG) snapshots prior to service selection and before agreement initialization. The paper proposes a filtering framework that is based on a faceted approach and that uses SLA templates to guide marketplace customers through available services. The framework design is presented along with the data-model of SLA templates. We report the results from testing the faceted filtering with two different SLA storage approaches and evaluate their appropriateness for the web application layer.**

***Keywords*-cloud marketplaces; SLA templates; SLA modularity; faceted filtering; document database.**

I. INTRODUCTION

A Service Level Agreement (SLA) accurately depicts how a service is going to be provisioned. Its explicit definition is necessary for both providers and consumers to measure and assess actual consumption of resources during service execution. The SLA description allows customers to have a clear idea before service commitment on how resources will be served. Hence SLAs can be helpful to make more informed investment decisions. Customers of service marketplaces can use SLA templates as *"What You See Is What You Get" (WYSIWIG) snapshots* when they navigate through available offers. We consider a SLA snapshot as a high level summary of a pre-agreed SLA.

Our discussion begins with the current role of SLAs in cloud marketplaces and with research challenges whose completion can advance the SLA utilization for IT services. The paper continues with the presentation of our filtering framework that uses the SLA template content to provide a multi-faceted navigation tool for customers. We position the framework within a service marketplace. A customer can filter views of available offers according to provisioning requirements. The goal of the faceted filtering is to gradually lead a customer to a reduced service offer list that is in accordance with customer requests, thus helping in the final service selection activity.

We describe the SLA template data model and the filtering framework design. Our analysis concentrates on how SLA information is stored and managed by a marketplace to help customers orient their navigation according to their provisioning requirements. To examine the applicability of the proposed framework, we simulate its operation using two different data storage approaches and evaluate their appropriateness for the web marketplace setting.

The paper is organized as follows: in Section II we formalize our problem setting and elaborate on SLA research challenges that we consider towards a large scale reality of efficient SLA content manipulation. Section III presents the data model for SLA templates, their construction process, the design of the filtering framework and the proposed database schemas. Section IV describes our experimentation and reports on preliminary results. Section V acknowledges related scientific work that tries to answer relevant research questions around SLA manipulation. The paper concludes with on-going work.

II. PROBLEM FORMALIZATION

A. SLA and SLA template role in cloud markets

In the following, the terms 'SLA template' and 'service offer' are used interchangeably. A Service Level Agreement (SLA) identifies the exact measurement and enables the auditing of described resource parameter values. The SLA definition provides an explicit view on how the provisioning of a service is planned. It also indicates precise bounds of service levels that a provider can supply.

Providers use SLAs during service execution to monitor service measurable attributes. Currently, SLAs hardly appear in cloud marketplaces. Promotion of IT offers to customers relies primarily on high-level service descriptions. The role of SLAs is peripheral and they are often materialized by documents of "terms-and-conditions" that typically do not involve functional service aspects.

In the literature, a SLA template represents a pre-instantiated agreement that is submitted by a service provider to a marketplace for customer consideration. The SLA template describes the agreement content that a provider is willing to accept during communications with customers. Thus a template describes precisely a provider's resource availability and provisioning plan. To decide which provisioning is more suitable for their needs, customers review SLA templates as service offers and proceed with either agreement initialization or negotiation with one or more providers.

 ISBN: 978-1-61208-271-4

We consider SLA templates as dynamic information that is updated at frequent time intervals. A marketplace or equivalently a service aggregator platform can use such templates as customer drivers for service selection since SLA templates enclose all details on how services are to be provisioned. SLA templates can be efficiently manipulated given that they follow a modular structure. Template content modularity allows viewing service offer sections as facets.

According to [10], a facet represents a category of ordered information data. It may contain flat or hierarchical information and can be divided into subcategories or sub-hierarchies. Moreover, a facet is described by attributes. In [10] the authors analyze how they have used hierarchical faceted categories (HFC) to organize the information structure of a navigational interface [5] for large data collections. Faceted navigation is a design pattern that enables flexible browsing through a web interface. Big market vendors have employed this pattern as it allows friendly navigation through multiple data hierarchies simultaneously. The ordering of information in multi-hierarchies makes the faceted-navigation pattern suitable to use with SLA templates also, since the native SLA structure is represented in the literature as a tree hierarchy [2], [9]. Figure 1 illustrates a high-level overview of the SLA schema proposed by [2].

In a SLA tree structure, facets represent SLA branches that describe ordered aspects of provisioning details. Representation of SLA facets can be combined with filters to facilitate the customization of facet attributes. In addition, filters generate new SLA facets by following selected traversing routes in the SLA tree path. Motivated by the faceted navigation pattern and its noticeable suitability with the SLA tree structure, we provide a framework that manipulates modular SLA templates to enable service customer navigation through available service offers.

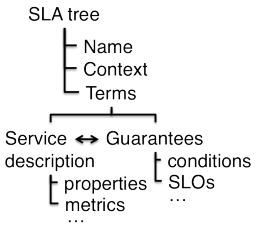

Figure 1. SLA tree structure according to WS-Agreement specification

Prior to service selection and agreement initialization, customers search through submitted offers to find services that match their business needs. The goal of SLA faceted filtering is to enable flexible service navigation that is driven by customer (either users or automated processes) provisioning requirements. The filtering process narrows down service offer views to only desired ones that fulfill requested provisioning parameters. A faceted navigation tool should provide filters that help customers indicate their service provisioning requirements according to existing offer availability. Filtered navigation facilitates rapid traversing between different offer views.

B. SLA manipulation challenges

In the scientific literature, SLAs are hardly viewed as end user documents, but merely as automated processes that assist the monitoring and scheduling of resources. In contrast, cloud marketplaces treat SLAs as static documents that do not allow for any processing. One challenge is to find the right equilibrium between these two orthogonal aspects and combine machine-readable with user friendly SLAs into a uniform process that can be used by both backend systems and front-end web services.

SLAs represent nested tree structures that include heterogeneous characteristics and are unbounded in terms of length and content. In the cloud business setting, diversified services are offered. Description characteristics and provisioning guarantees vary considerably, even if they describe similar services in different contexts. Providers from different business domains use customized terminology to describe service parameters, metric functions and guarantee definitions. Terms like ”availability”, ”throughput” or ”performance” are usually included in ambiguous ways in service descriptions, which may be confusing for service customers. Various vocabularies of provisioning terms represent a primary cause for SLA heterogeneity. On a wide scale, SLA semantic and structural heterogeneity represents a challenge because it complicates SLA template comparison thus hindering any attempt to efficiently manipulate SLAs in open marketplaces.

SLA formulation highly depends on resource availability. Hence to manipulate SLA templates for customer interest, we need to first ensure that the template content can be updated dynamically. As the SLA depth is unbounded, frequent updates may cause performance delays in the information exchange between customers and providers. Thus, the storage schema of SLA templates represents a challenge. On one hand, one may argue that since SLA templates represent dynamic information objects, they should not be stored at all. Instead, they should be kept in-memory for as long as they are valid and then be immediately replaced. On the other hand, a modular SLA data model allows to persist SLA templates for longer time periods and to run frequent content updates according to provider resource capacity and provisioning availability. In this paper we work on the latter aspect.

Viewed as a tree hierarchy, the SLA content may include nested branching, which may lead to alternative information content. A challenge for the manipulation of SLA templates is to select a content structure that facilitates quick traversing within nested information routes. A modular structure provides independence between inner SLA components thus helping the exploitation of finer grained information. Mod-

 ISBN: 978-1-61208-271-4

ularity allows for categorization of SLA parts and indicates data management structures that may apply for diverse types of SLA formats.

SLA content heterogeneity addresses issues that deal with scientific opportunities for data management, information retrieval and language processing research. In addition, it highlights the need for SLA standardization. Currently, SLA formalization is not supported by a standard to allow classification of key performance indicators (KPIs) or to mandate inclusion of specific functions per business domain.

The scientific computing community has primarily used the WSLA [9] and WS-Agreement [2] language specifications to express SLAs. The GRAAP working group proposed the WS-Agreement specification [2] as a language and a protocol to conduct SLAs. The WSLA [9] language specification has been proposed by IBM research on utility computing. Both approaches denote SLA language semantics in XML notation. According to [9], a SLA complements a service description. Moreover, both specifications suggest the use of customer and provider templates for the exchange of counter offers in the process of agreeing on service levels.

III. Filtered navigation and template repository

We propose a filtering process that is different from direct comparison and exchange of SLA templates. The process uses provider templates to construct filters, based on which customers express their provisioning preferences. The outcome of the filtering process does not represent the final selection decision of a customer, but rather a subset of available service offers that satisfy the imposed filters.

We assume homogeneity of template structure with respect to the ordering of sections and terms as proposed by [2]. Filters are created according to SLA facets. The following paragraphs describe the SLA template construction, the design of the filtering framework and the SLA template storage schema.

A. SLA template construction

[2], [9] propose that a SLA consists of three primary sections:

(i) service description,
(ii) guarantees or obligations and
(iii) an informative section regarding involved parties and/or the provisioned service

[2] names the latter section SLA context. To construct SLA templates, we follow the WS-Agreement guidelines, but express the template content in JSON [8] notation. The SLA template construction steps can be summarized as following:

1) Parse XML sample into JSON
2) Use (1) to create SLA template data model
3) Create database schema according to (2)
4) Retrieve service descriptions from marketplace
5) Order data from (4) into service types
6) Create fictional information, order according to (5)
7) Shuffle information from (5) and (6) into randomly generated data lists
8) Load (7) into CSV files
9) Load (8) into database

The native WS-Agreement format comes in XML notation. Hence we initially parse a WS-Agreement template sample from XML to JSON. We use the JSON sample to create the data model for our SLA templates. From the native WS-Agreement specification, we employ the proposed sections of guarantees, description terms and agreement context, but order them accordingly to address the filtering need for modularity. Moreover, we extend the context section that we refer to as non-obligation attributes, and add information regarding the provider infrastructure and the customer data-storage location. We keep the service description joined with associated metrics and guarantees. Furthermore, we add a separate section for guarantees that apply to the overall service and that may or may not be measurable. We use this section to include customer monitoring options and provider obligations that indicate QoS bounds, e.g., service helpdesk availability. Customers typically need to be aware of such options before service commitment. Figure 2 illustrates the deduced SLA data model that we use to create the database schema for our templates.

The proposed SLA data model exploits information granularity by categorizing data into distinct SLA modules. This ordering allows for isolation of internal SLA root components, without depriving their inner depth in terms of nested branching. Nesting within a SLA template module depends on the information content. For example, non-obligation terms do not contain additional branches and remain consistent for all templates, regardless of service type. Service description and associated guarantees expand to multiple branches. The depth-level of nesting is of interest as it affects the template storage schema and hence the filtering flow process. Moreover, the suggested data model allows expanding the SLA content into distinct themes. Figure 2 depicts SLA data modularity with the letter N to indicate granularity of themes.

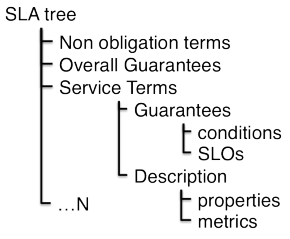

Figure 2. SLA data-model

Following, we retrieve information of service descriptions from the Amazon WS marketplace [1]. In particular, we derive service profiles that relate to storage, network and virtual machines. We order this information into nested

lists according to service type. Wherever necessary, we reformulate the retrieved data and complete them with fictitious information to cover the content of our SLA template.

We iteratively call a Python process to generate lists whose elements are assigned randomly from the classification of ordered data. We intentionally apply variations in the nesting depth of lists. Generated data are not skewed. Still, we aknowledge that in real market conditions services do not share the same level of popularity, thus customer preference. The Python process loads the generated data lists into comma separated value (CSV) files and from there to the database management system (DBMS) in use.

In this manner we create template sets for all three derived service profiles. Generated templates follow the proposed SLA data model, but differ in depth level and content. The template construction procedure simulates provider submissions of service offers into a cloud marketplace.

B. Filtered navigation through service offers

Figure 3 illustrates our proposed filtering framework. The framework design consists of two layers. One tier depicts all possible combinations of cloud stack layer [4], service type and offer validity as a three dimensional Cartesian coordinate system. The other tier presents the proposed SLA tree as a cube, where cubic sides indicate root-facets of filtering and service offer views. The cube selection is indicative of the proposed data model because a template may contain up to n SLA root-themes. The multidimensional structure depicts the inner-depth volume and interconnections of nested SLA information.

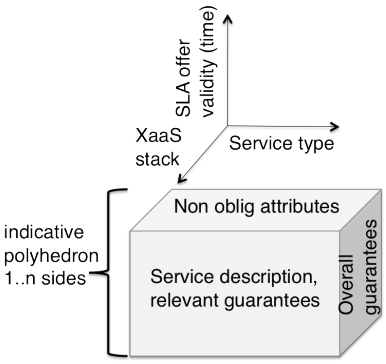

Figure 3. SLA filtering framework

Parameter combinations from the two filtering tiers indicate navigation and filtering options. For the suggested cubic representation, we point out the following entry-points:

(a) Select a combination of cloud stack layer, service type and time interval. We take into account interdependencies that may exist between cloud stack layer and service type, since several services are mapped to a single cloud layer.
(b) Select (a) and combine with filtering of non-obligation attributes.
(c) Filter only non-obligation attributes.
(d) Select (a) and combine with filtering of high-level service description terms.
(e) Select (a) and combine with filtering of overall guarantee parameters.
(f) Filter only overall guarantees.

The selection of entry-points designates one or more conditional queries that are processed transparently from customer actions, on the backend. The instantly returned result facilitates further navigation from a refined subset of existing offers, where deeper-level filtering options are provided. The navigation process gradually leads to a minimal set of preferred service offers that satisfy provisioning requirements according to submission of customer parameters. The method can be also deployed as an incremental process, where the system keeps track of customer selections on each step and accordingly regulates the flow of results.

The inherent modularity of the proposed SLA data model and its representation as a multidimensional structure allows for quick and selective navigation through designated nested information. At any point the navigation route can change by either selecting a different combination of SLA facet or by re-arranging filter values. The approach provides flexibility to navigate through available service offers from the provisioning aspect that a customer is mostly interested in.

Thus a customer may directly navigate through service profile attributes and associated provisioning guarantees by selecting the type of service and by filtering initial parameters from the description category. Alternatively, a customer may first look into non-obligation attributes if, for example, there is a provider or a data-location preference. Moreover, a customer may simply search for particular guarantee attributes that are irrelevant of service type.

Entry navigation and filtering points can be extended accordingly to the SLA structure branches or respectively SLA facets. Special facets can be introduced to illustrate provisioning guarantees that deal with service provider and customer concerns about, for example, energy efficiency or environmental impact.

C. SLA template storage

Filters in faceted navigation translate customer choices into conditional queries. In this work, we consider and experiment the faceted filtering with two different data management approaches.

In one case, SLA templates are stored and manipulated in a relational DBMS. Service offer information is kept into distinct tables and at a granular level of detail according to the template data model. In [2] a unique identifier (uid), located into the SLA context section, accompanies every SLA template. To resemble this relationship in the relational

 ISBN: 978-1-61208-271-4

database schema, we set a uid as the primary key (PK) of the non-obligation attributes table. Similar to [2], this PK acts as a reference key for the identification and matching of any incoming template. Moreover, the uid serves as foreign key (FK) to service description and overall guarantees tables that are associated with a specific template instance. This relationship resembles the native SLA tree structure of [2]. Figure 4 shows the relational database schema of our design.

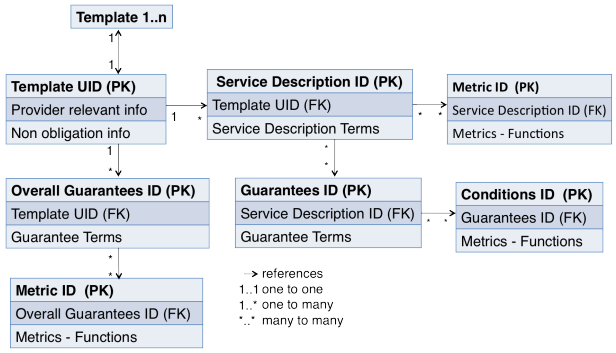

Figure 4. SLA template - relational database schema

To illustrate inner content branching, the PK of the service description table acts as a FK to associated rows of metric and guarantee tables. Similarly, the overall guarantees table is associated to metrics and function definitions for the measurement of referenced guarantees. In this order, the relational schema offers an alternative to the native XML structure proposed by [2]. This database design achieves the necessary granularity in terms of parameter details to allow for conditional queries on term and metric values.

We express and manipulate SLA templates using the JSON format. This choice was driven by our objective to test faceted filtering with a structured query language of a relational database and with a NoSQL data processing system, which in this work represents a document database. Since SLAs are machine-readable documents, a NoSQL DBMS may prove suitable for the marketplace scenario that typically operates over HTTP.

The document database design follows a nested dictionary structure. Compared to the relational schema, document collections represent tables and respectively documents represent records (table rows). The database design looks a lot like the relational one, but SLA templates are stored as nested documents. Although, schema conformance is not a pre-requisite for a document database, every stored document follows a generic SLA template structure and accurately corresponds to the information stored in the relational database. Figure 5 illustrates the NoSQL schema design.

Every stored document is accompanied by a unique identifier and embeds dictionaries (or sub-documents) to map FK

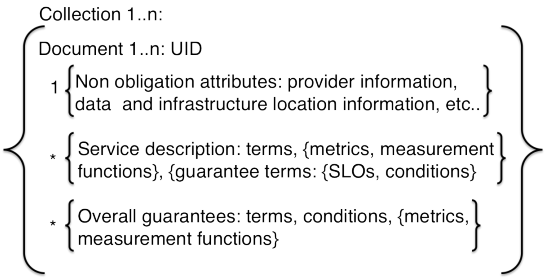

Figure 5. SLA template - document database schema

relationships from the relational database to the document schema. Each document contains one dictionary that holds non-obligation service attributes and one or more dictionaries to present service description parameters and overall guarantees. Similar to the relational model, description terms and overall guarantees enclose associated service attributes and respectively metrics, which in turn embed additional nested information.

IV. Experimentation

A. Filtering simulation setup

We simulate the faceted filtering operation using the entry navigation-points analyzed in Section III. We assume that an IT marketplace provides SLA faceted navigation as an interaction tool for customers to submit their criteria and guide their browsing of service offers through provisioning requirements. We emulate customer instances and the SLA faceted filtering in a client - server architecture. Our goal is to measure the server response time to incoming customer requests and the scalability of the filtering operation as the number of simultaneous requests increase.

We setup the simulation environment on a 24-processor computing machine. The model of each processor is Intel Xeon and every processor runs at 2.50 GHz. The computing machine includes 128GB of RAM and operates on Ubuntu 12.04, Linux version 3.2.0. We deploy the Tornado web server [14] that is natively written in Python, to represent the server side of the simulated environment.

Filtering is accomplished by simultaneous processing of queries and our tests target the parallel handling of client requests. We prepare multithreaded Python scripts that use data from SLA facet attributes, generate random parameter values and pass them as HTTP GET requests to the web server. Randomly generated parameters simulate the customer filtering input. We keep the values of generated parameters within the value range of existing SLOs and description terms. This configuration does not guarantee that customer requests are always satisfied, because every incoming request submits a diverse number of SLA requirement values, whose combinations may not map to an existing SLA offer.

 ISBN: 978-1-61208-271-4

In every run the server receives parameter values from each incoming HTTP request, generates a conditional statement and sends it for processing to a DBMS. A server process reads the returned result set from the DBMS and updates the customer view with matching available offers. The server handles client requests with the help of common gateway interface (CGI) Python scripts, which are multi-threaded to assist the concurrent request serving.

SLA templates are submitted and updated on-demand, transparently from customer activities. Section III describes the process of creating our SLA templates and loading their content into a DBMS. In our simulation the marketplace uses a centralized data repository for the SLA template storage. Our datasets are derived from the stored template content.

We use MySQL DBMS [13] for the relational database and MongoDB [12] for the document database. Both DBMS are deployed on the same machine as Tornado to reduce TCP communication overhead. Measurements are derived from testing with each database separately. Each table in the MySQL database is loaded with approximately 150,000 records. In MongoDB this number amounts to 35,000 documents with an average document size of 1289.44 kb.

B. Experimental setup

The experimentation simulates the process of sending and handling concurrent client requests and returning the results over HTTP. The entry points that we introduced in Section III designate the main use cases of our testing. Every entry point represents a number of query parameter values that are passed to the server and from there to the respective database. Incoming parameters represent SLA facet attributes. Their number depends from the facet type and its nesting depth. We range incoming submissions between 2, 10 and 20 parameters.

We start from the upper, more generic, tier of the filtering framework (Figure 3) and submit 2 parameters to represent an initial choice of service type and offer expiration time. We gradually combine filtering attributes from both framework layers and reach nested template information. Our testing deals with different customer use cases. We simulate the case, where a customer has a provider and a data storage location preference and hence filters only attributes of the non-obligation facet, which in our case represents a submission of 5 up to 10 parameters.

We also consider a customer, who wants to look into service offers with specific description characteristics and explicit guarantee values. The customer selects a desired service type and filters attributes of the service description facet. Submission to the server ranges from 10 up to 20 parameters. We use the same parameter range to deal with the submission of overall guarantee criteria and to combine filtering attributes from different SLA facets.

We run the same number of experiments for both databases and categorize them in three test suites. In the first test set we measure the total time of the faceted filtering operation over HTTP. The total time starts from the point a client request reaches the server up to the point the server returns the result to the client. Timings include HTTP and backend processing overhead.

The second test suite includes faceted filtering runs that are processed locally on the machine where the server and the two databases reside to avoid additional network overhead. The third test suite is also based on local runs, but measurements combine the query processing from filtering and database updates. We prepare an extra set of update statements for both databases to measure their potential overhead on the filtering operation. In each run update queries are processed in parallel to filtering requests and account for an extra 10% of workload on the total database processing. Local communication between server and DBMS is achieved via Unix sockets for the MySQL database and over localhost for MongoDB.

Queries in each DBMS are similar in terms of number and type of conditions, but the values of conditional parameters are randomly generated for every query. For the MySQL case, conditional queries take the form of SELECT statements, where the number of conditions varies according to the incoming parameters. For MongoDB queries are represented in binary JSON (BSON) format. The MongoDB alternative for SELECT statements is the formation of queries with the find() method. For every filtering point, we repeat the same test for 10 runs and take the average time from their accumulation. We also gradually increase the number of submitted HTTP requests. We begin with 100 simultaneous requests and reach up to 100,000 concurrent requests for both MySQL and MongoDB.

C. Observations and evaluation of results

The graph in Figure 6 shows the results for both MySQL and MongoDB from running with 2, 10 and 20 requested parameters over HTTP. The y-axis represents the average total time for each performed run and the x-axis indicates the gradually increased number of incoming requests that the web server receives. The average time is close to constant for both MySQL and MongoDB. Derived curves for all runs are fitted to highlight the small range of fluctuactions in the query processing results.

The filtering operation over HTTP takes approximately 1.87 seconds less for queries that are processed in MongoDB compared to the average time in MySQL. This approximate time difference prevails for all HTTP runs regardless of the number of incoming requests or the number of simultaneously processed queries. The difference can be justified by the fact that when MongoDB retrieves a document from a collection, the whole document is loaded into memory along with any embedded dictionaries. Thus, retrieving information from any nested dictionary comes at a minimal cost as soon as the root document is loaded into memory.

 ISBN: 978-1-61208-271-4

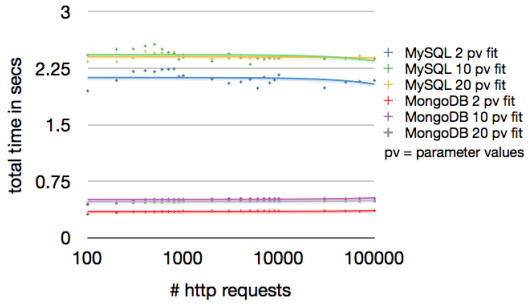

Figure 6. Average total filtering processing time over HTTP: MySQL and MongoDB

The embedding feature of MongoDB provides an alternative for MySQL JOINS [12].

Figures 7 and 8 present the results from the locally executed test sets. The average query processing time for MySQL is illustrated in Figure 7 and for MongoDB in Figure 8. For both graphs the y-axis represents the average query processing time in seconds and the x-axis the gradual increase in the number of submitted queries.

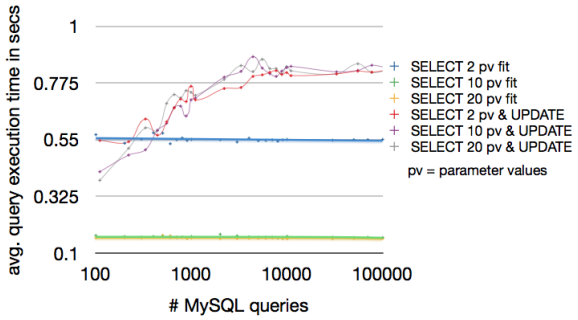

Figure 7. Average query processing time MySQL via UNIX sockets

The average query processing time for local faceted filtering (both SELECT and find() statements) is almost constant and in fact identical (0.16 seconds average) for both DBMS. The only exception is the MySQL SELECT query with 2 conditions, where the average time is nearly 0.38 seconds more than the SELECT queries with 10, 20 conditions and the respective find() statements in MongoDB. In both graphs, the curves that illustrate the average query processing time of the local faceted filtering are fitted to designate the small fluctuation range of the result set.

Figures 7 and 8 also illustrate the results from local runs that combine updates and faceted filtering. For both DBMS, updates are executed in randomly selected tables (respectively collections) and with randomly generated conditions. Updates affect multiple records of one or more tables (collections) but not those, where the SELECT/find() query operates. For both databases the results from the mixed

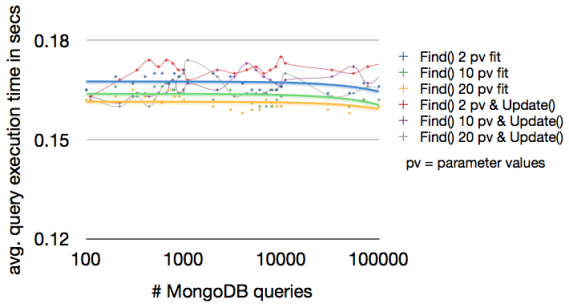

Figure 8. Average query execution time MongoDB over localhost

processing are not linear due to the random factor that affects the volume of updates. Compared to local faceted filtering, the cost of the update operation for MongoDB is negligible. For MySQL the cost is nearly constant at 0.57 seconds, with the exception of the SELECT statement with 2 conditions.

Our overall testing indicates that possibly a NoSQL approach like the MongoDB DBMS fits better for the web scenario, where SLA offers are manipulated over HTTP. In the local running mode, both DBMS share comparable performance. Still, the combination of updates with SELECT statements appears to be more expensive for MySQL than for MongoDB.

V. Related work

In [3], the authors propose an approach for automated matching of customer and provider templates by discovering semantically equivalent SLA parameters. The authors highlight that the absence of SLA standardization inevitably leads to variations in the definition of semantically related terms. They use a machine learning methodology to illustrate their matching comparison. The authors assume the existence of a knowledge repository that is responsible for managing incoming SLA templates and template mappings. As their work is focused on the comparison of SLA terms from diverse templates, the authors do not go into detail about the repository structure or the exposure of SLA parameters through a web interface.

In [11], a decision-support framework is proposed to assist the selection of infrastructure resources and the migration of services from local to virtual platforms. Although the approach is not explicitly directed towards SLA manipulation, the decision-support operation uses service attributes that are derived from provisioning parameters. The authors do not deal with customer navigation in a marketplace, but assume submission of service requirements by potential customers. Service attributes are structured in hierarchies. The authors apply the decision-support framework into a realistic use case to prototype the filtering of customer requirements on available service parameters. Still, they do not elaborate on how retrieved information is either stored or managed.

 ISBN: 978-1-61208-271-4

The work described in [6], [7] is a motivating schema for SLA-aware service-oriented infrastructures. In the proposed architecture, customer-provider interaction takes place over a service registry. The model can be extended to current conditions of service provisioning. SLA templates in the form of service offers are included in marketplaces and customers select services according to their provisioning preferences. A marketplace can then expose SLA offers in the same way a registry exposes service descriptions.

VI. Conclusions & on-going work

The scope of the presented work has been to promote SLA aspects from post-agreement monitoring instruments to pre-agreement manipulation objects. SLA templates represent pre-initialized agreements and describe provisioning plans of service providers. We presented our SLA data model that assumes structure homogeneity and is based on the WS-Agreement language specification. Our data model supports modularity of internal components as this feature enables the extraction of SLA facets by categorizing information into distinct themes.

We described how we constructed SLA templates and used them with a faceted filtering framework that enables customers to browse through available services according to their provisioning requirements. Service customers utilize facet attributes as filters to express their objectives and to get views of preferred provisioning arrangements. We demonstrated use cases of filtering according to facet preferences that customers would like to be aware of before service commitment. The approach can be extended to include additional filtering criteria that influence provisioning expectations and are derived from non-SLA related objectives (e.g., risk, security, energy efficiency).

For the filtering experimentation we used two different DBMS approaches, a relational one represented by MySQL and a document one represented by MongoDB. We assumed that customer requests arrive concurrently and need to be served immediately. Both databases share their tradeoffs. MySQL is a seasoned DBMS, possibly suitable for back-end processing of SLA data. MongoDB represents a new product that appears to more efficient in terms of query processing time on web operations. Our results indicate that the MongoDB approach seems more suitable for SLA manipulation on the HTTP layer, where client requests reach the web server in large-scale mode and need to be handled simultaneously.

We continue the refinement of the SLA data model and the filtering framework experimentation with alternate modes of template persistence. An alternative to the NoSQL document approach is a database system that supports the Resource Description Framework (RDF) data structure. RDF encoding enables the representation of information in a graph form, where connections between nodes indicate semantic relationships. This attribute is of particular interest for SLAs, as it supports the classification of SLA modules and promotes the creation of semantic vocabularies that can be associated in a distributed sharing mode.

Our next challenge is to extend the filtering framework into a recommendation mechanism that provides customer-tailored SLA suggestions by using a given user profile. The filtering framework can be considered as a pre-requisite of the recommendation system because it provides a tool to keep track of customer navigation behavior and filtering preferences.

Acknowledgement

This work is supported by the Swiss National Science Foundation (SNSF), grant number 200021E-136316/1

References

[1] "Amazon WS Marketplace," accessed Sep. 2012, https://aws.amazon.com/marketplace.

[2] A. Andrieux *et al.*, "Web Services Agreement Specification (WS-Agreement)," retrieved Oct. 2011, from http://www.ogf.org/documents/GFD.192.pdf, Open Grid Forum, Grid Resource Allocation Agreement Protocol (GRAAP) Working Group.

[3] C.Redl, I.Breskovic, I.Brandic, and S.Dustdar, "Automatic SLA Matching and Provider Selection in Grid and Cloud Computing Markets," in *Proc. of the 2012 ACM/IEEE 13th International Conference on Grid Computing (GRID '12)*. IEEE Computer Society, 2012, pp. 85–94.

[4] Fang Liu *et al.*, "SP 500-292 Cloud Computing Reference Architecture," retrieved Oct. 2011, from http://www.nist.gov/manuscript-publication-search.cfm?pub_id=909505, National Institute of Standards and Technology (NIST), Sep 2011.

[5] "The Flamenco Search Interface Project," accessed Oct. 2012, http://flamenco.berkeley.edu.

[6] H.Ludwig, "WS-Agreement Concepts and Use of Agreement-Based Service-Oriented Architectures," IBM Research, Tech. Rep., 2006.

[7] H.Ludwig, A.Dan, and R.Kearney, "Cremona: an architecture and library for creation and monitoring of WS-agreements," in *Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC '04)*. ACM, 2004, pp. 65–74.

[8] "JavaScript Object Notation," accessed Aug. 2012, http://www.json.org.

[9] H. Ludwig, A. Keller, A. Dan, R. King, and R. Franck, "Web Service Level Agreement (WSLA) Language Specification," retrieved Oct. 2011, from http://www.research.ibm.com/, IBM Corporation, Tech. Rep., Jan 2003.

[10] M.A.Hearst, "Clustering versus faceted categories for information exploration," *Commun. ACM*, vol. 49, no. 4, pp. 59–61, Apr. 2006.

 ISBN: 978-1-61208-271-4

[11] M.Menzel and R.Ranjan, "CloudGenius: decision support for web server cloud migration," in *Proceedings of the 21st international conference on World Wide Web (WWW '12)*. ACM, 2012, pp. 979–988.

[12] "MongoDB manual," accessed Sep. 2012, http://docs.mongodb.org/manual.

[13] "MySQL," accessed Sep. 2006, http://www.mysql.com.

[14] "Tornado web server," retrieved Sep. 2012, from http://www.tornadoweb.org.

 ISBN: 978-1-61208-271-4