Robust Content-Centric SLA Enforcement in Federated Cloud Environments

Nikoletta Mavrogeorgi(✉), Athanasios Voulodimos, Vassilios Alexandrou, Spyridon Gogouvitis, and Theodora Varvarigou

School of Electrical and Computer Engineering, National Technical University of Athens, Athens, Greece
{nikimav, thanosv, alexv, spyrosg}@mail.ntua.gr, dora@telecom.ntua.gr

Abstract. In this paper we present a system for declaring and enforcing SLAs in Cloud environments. The SLAs proposed are enriched with content terms, storlets and federation capabilities and provide high degrees of customizability for clients. A mechanism for SLA enforcement has been designed and implemented which, based on policies, measurements, usage data computations, and monitoring methods permits proactive SLA violation detection and handling. SLA renegotiation is supported as well. The proposed framework has been developed and evaluated in challenging scenarios in a variety of different application domains.

Keywords: SLA schema · SLA enforcement · Monitoring · Proactive SLA violation detection · Content centric storage · Storlets · Federation · Evaluation

1 Introduction

The cloud paradigm has undoubtedly revolutionized the IT landscape given the significance of on-demand cloud services offered by increasingly powerful cloud providers. The need for secure storage and retrieval of data is shared among all types of clients who can range from a simple user who uses a cloud service to store their photos to big enterprises (in a variety of sectors, i.e. healthcare, media, IT, banking, industry, etc.), which store financial and other sensitive data. A Service-level agreement (SLA) is a contract between a (cloud) service provider and a customer that specifies, in measurable terms, what services the provider offers. A SLA often includes metrics that specify the performance, availability, and security assured to the customer, as well as penalties for violating these requirements.

SLAs in Cloud architectures have been the focus of attention of a significant number of researchers and professionals, especially with the rapid adoption of cloud based solutions in many different application domains. SLA schemas XML schemas that represent the content of an SLA. Some existing approaches for SLA schemas and the corresponding languages to define service description terms are: SLAng [1], WS-Agreement [2], WSLA [3], WSOL [4], and SWAPS [5]. Nevertheless, the proposed schemas have limitations. SWAPS is quite complex and the implementation is

© Springer International Publishing AG 2017
J.Á. Bañares et al. (Eds.): GECON 2016, LNCS 10382, pp. 35–48, 2017.
DOI: 10.1007/978-3-319-61920-0_3

not publicly available. WSLA and SLAng have not further development at least since 2009. Apart from this, SLAng does not permit to define management information such as financial terms and WSLA has not formal definition of metrics semantics. WSOL lacks SLA related functionalities, such as the capture of the relationship between service provider and infrastructure provider. The WS Agreement is a Web Services protocol for establishing agreement between two parties using an extensible XML language for specifying the nature of the agreement, and agreement templates to facilitate discovery of compatible agreement parties. It allows arbitrary term languages to be plugged-in for creating domain-specific service description terms.

Two challenging research issues are the requirements translation from high level metrics to low level requirements and vice versa and the proactive violation detection. Several proposals have been made for these issues, but very little for cloud environments. For instance, GRIA SLAs [6] suggest a solution for avoiding violations but concerns only Grid environments. The LoM2HiS framework [7] proposes the translation of low level metrics to high level terms that are used in Cloud SLAs, but not the reverse translation. Also, they are based on generic characteristics and terms (e.g. availability) which are not application specific. The LAYSI framework [8] supports two kinds of monitors sensors, the host monitor and the runtime monitor sensor. The latter senses future SLA violation threats based on resource usage experiences and predefined threat thresholds. In DesVi [9], an architecture is proposed for preventing SLA violations based on knowledge database and case-based reasoning. It also uses the LoM2HiS framework for the requirements translation.

In [10] an analysis of SLA violations in a production SaaS platform is described, while [11] presents a scalable, stochastic model-driven an interacting Markov chain based approach to quantify the availability of a large-scale IaaS cloud. In [12] the authors present an aggregation mechanism for merging service-level objectives and for guaranteeing a single SLA that specifies obligations and responsibilities of all participants in a federation. The framework in [13] uses a portfolio-based optimisation to improve SLA compliance by diversifying the selection and consequently the allocation of traded instances of web services from multiple providers. An end-to-end framework for consumer-centric SLA management of cloud-hosted databases is proposed in [14] to facilitate adaptive and dynamic provisioning of the database tier of the software applications based on application-defined policies for satisfying their own SLA performance requirements. SALMonADA [15] performs an automated monitoring configuration and it analyses highly expressive SLAs by means of a constraint satisfaction problems based technique. In [16] a new proactive resource allocation approach is proposed aiming at decreasing impact of SLA violations by using two user's hidden characteristics, i.e. willingness to pay for service and willingness to pay for certainty. In [17] an SLA implementation for Cloud services based on the CMAC (Condition Monitoring on A Cloud) platform is proposed, while in [20] decision-making with regard to availability SLAs is explored.

In this paper, we present a system for declaring and enforcing SLAs in Cloud environments where commitments for using Cloud services are defined. The SLAs are enriched with content terms, storlets and federation capabilities. Additionally, many SLOs (Service Level Objectives) are supported at different levels permitting the clients to have customized SLAs. A mechanism for SLA enforcement has been designed and

implemented which, based on policies, measurements and usage data computations, permits SLA violations to be handled and imminent SLA violations to be proactively detected. Moreover, renegotiation is supported, i.e. a client can change their SLA for a variety of reasons. The system permits the SLA renegotiation and based on measurements and usage data can suggest changes to the existing SLA aiming at being more compatible to its data usage or for cost reduction. The main core of the presented framework was developed in the context of the VISION Cloud EU project [18].

2 Cloud Models and Enriched SLA

Our proposed system permits the storage of objects and their retrieval anytime and from anywhere. In order this to be achieved replicas of the objects are stored in appropriate locations. The clients of the Cloud are the tenants and their users. A tenant is the unit that subscribes to storage cloud services. A tenant defines its users. The users handle the objects that are stored in the Cloud. The objects are stored grouped by containers. Containers serve as an aggregation point for grouping related data objects together. Policies can be set on a container basis and are applied to all of the objects in the container. A container is associated with an SLA and based on it the number and the locations of the replicas of the objects are defined. The Cloud is composed of multiple data centers. Each data center is split to clusters and each cluster consists of multiple nodes which contains the servers. This hierarchy is stored in Catalogs. Each level has its own aggregated catalog and uses GPFS-SNC. A local catalog enables mapping objects to file paths and the node containing the object. The resource model can be seen in Fig. 1. The existence of a resource model in the PaaS layer of a Cloud environment serves the need of management in terms of resource allocation, services deployment and execution and finally optimization.

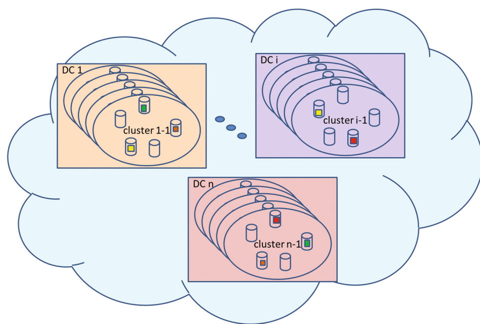

Fig. 1. Resource model.

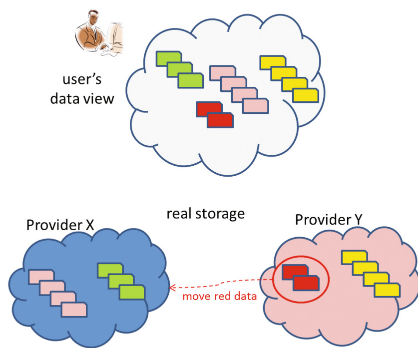

Fig. 2. Federation.

The proposed SLA contains additions that are significant for Cloud provisions. Apart from the classical data of an SLA such as availability levels and responsibilities

data between the provider and the tenant, our SLA is based on content terms and contains more SLO's in order to be customized as per the tenants' desires. Also, it contains federation and storlets support and commitments. Finally, it provides renegotiation with suggestions for better SLA according to the tenant's usage.

Content term addition. The innovation of declaring content terms permit the Cloud to provide content-centric services. Also, it links content with performance estimates, decisions for moving computation close to storage, pricing models, etc. Another advantage is that support more efficient capabilities, e.g. quicker search and retrieval of the objects, services regarding the content term for instance in case of media, high performance video related services are supported.

SLOs. The SLOs are not restricted in availability. There are many requirements in different levels that are selected by the tenant according to their desires. A tenant can balance the cost with the supported levels. For instance a tenant that uses data related to media can demand throughput of the highest level for the objects that concern the daily news as these are highly demanded whereas to choose lower throughput in order to have less cost. Finally, geographic constraints are supported. The tenant can choose the desired regions and black list regions where restricts its data to be stored.

Storlets. Storlets are executables which provide capabilities for supporting and improving the services that are offered to the tenants. Some storlets are: data compression, file transformation in various formats, translation, speech2text, text2speech, text2pdf, pdf2txt, transcode format, classify photos, extract data for patient, etc. Each storlet has a condition and an action that is executed when the condition is met.

Federation. Cloud federation is the practice of interconnecting the cloud computing environments of two or more service providers for the purpose of load balancing traffic and accommodating spikes in demand. Also, there is a need for interoperability that is to move data between providers without this to be visible to data usage. There are many reasons for having data in more than one Cloud providers. For instance, the SLA (or application) requires services that cannot be found on only one provider, the amount of resources required goes beyond what a single provider can offer, or the required performance cannot be guaranteed by any single provider alone. In our system we support change of storage providers without data lock-in and single view of storage across multiple providers. The SLAs federation section declares if the federation is permitted and with which providers. Figure 2 shows the federation view. The tenant can have access to their data without being interested in which Cloud provider they are stored and without knowing the process of data movement between providers.

Requirements. The requirements addressed were chosen based on use cases of VISION Cloud project and include: Throughput (some requirements require specific throughput levels aggregated at the level of tenant, whereas others require throughput per request), Durability (asked for specific durability levels), Availability, Duration (constraints regarding the duration of the requests: latency and response time), Security and privacy (Authorization, authentication and guarantee of proper use), Geographic constraints (User determines in which regions he desires to store his data), Violations checks (All the requirements should be checked and met during the SLA lifecycle),

Storlets (SLAs also provide storlet selection), CDMI (The external interfaces should be CDMI compliant), Billing, CRUD operations.

3 SLA Management and Enforcement

Our proposed SLA Management provides a robust end-to-end SLA system starting from the SLA negotiation and runtime enforcing the currents SLAs with the agreed services and commitments. The SLA enforcement apart from dealing with providing the requested services according to the agreed SLOs with the tenant, is responsible for checking for SLA violations and for detecting proactively imminent SLA violations. In case of possible SLA violations it tries to avoid them with corrective actions. The SLA enforcement is based on policies and usage data analytics that are checked based on monitoring data.

As far as the architecture is concerned, there are two main components: (a) the **SLA Negotiator** which handles the services that have to do with the external communication with the clients that is SLA negotiation, SLA renegotiation, SLA templates generation, billing etc. (see Fig. 3) and (b) the **SLA Enforcer** which deals with the enforcement of the SLA and handles services that are needed in the Cloud internally such as policies generation for monitoring and analysis, check of SLA violations, proactive detection of possible SLA violations and decisions for corrective actions (see Fig. 4). The SLA schema that is used is described in detail in [19].

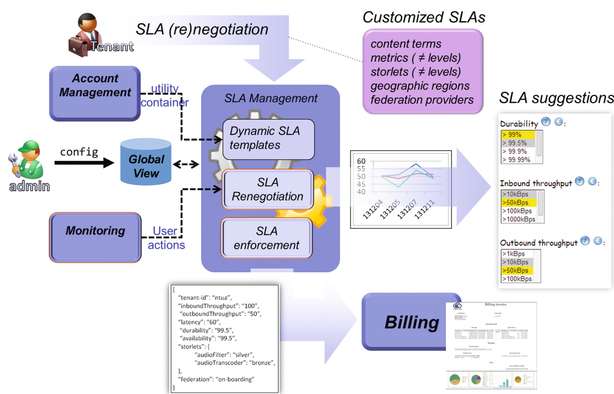

Fig. 3. SLA negotiator.

SLA Negotiator. The SLA Negotiator component is responsible for implementing the external interface of the platform with regard to SLAs.

SLA Negotiation. The negotiation of SLAs is realized by taking into account content related terms, thus reflecting the content centric approach of VISION Cloud. Different capabilities and costs are provided depending on the selected content term. To sign an SLA, a tenant chooses and fills an SLA template. SLA templates are generated

dynamically according to the supported capabilities of VISION Cloud and the content terms. SLA templates contain the supported metrics with different levels, by which the SLOs are derived when the tenant chooses his desired level, the supported services, the obligations of both parties (provider and tenant), the billing rules and the penalties in case the agreed service level is not met. An SLA template also contains terms related to the federation capabilities of the platform, which define the tenant's ability to perform federation and with which providers. Additionally, a section for storlets is provided. The set of storlets available to the tenant is based on the tenant authorization and the type of the requested SLA template. Some storlets in the set might be compulsory, and some optional; besides these storlets, other optional ones are displayed and can be additionally chosen.

SLA Renegotiation. SLA renegotiation is supported. During SLA negotiation there is a section for determining which data the tenant can request to the provider to be modified during the SLA lifecycle (e.g. levels of the permitted SLO, addition of storlets, federation permissions, etc.). If both parties agree, the SLA is changed, and it is pushed to the internal system with all the necessary modifications. Also, suggestions are provided to the tenant informing of what terms may be changed in his SLA in order to better suit him (e.g. adding storlets, or using a lower level SLA which already covers his needs at a lower cost). The SLA Management component is easily modifiable, and contains an automatic way to handle changes in the SLA schema.

Reports and SLA Data. SLA Management provides a user friendly GUI, by which the user can request the creation of an SLA and can delete, edit, view an SLA or a list of SLAs that concerns the current user. Also, the user can be notified about certain events, receive reports, and view older notifications and reports (e.g. SLA violations occurred in a certain period). The GUI supports user authentication in order to enforce security and privacy policies.

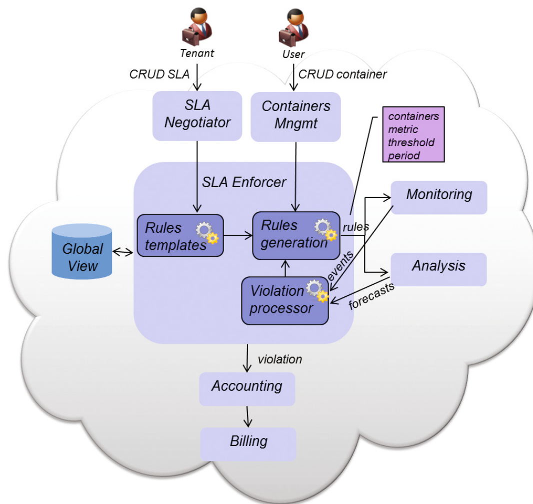

Fig. 4. SLA enforcer.

SLA Enforcer. The SLA Enforcer deals with the system configuration regarding the SLAs and the enforcement and maintenance of SLAs between tenants and providers.

Container Configuration. During container creation, the SLA Enforcer is contacted by the Container Management component to obtain the QoS requirements stemming from the chosen SLA. One responsibility is to translate the requirements translation from the high-level QoS metrics to the low-level ones and vice-versa. Translation from the high-level metrics specified in an SLA (e.g. durability) to low-level metrics by which the internal system works (e.g. number of replicas) is important during SLA management. It is also needed for checking the feasibility of the requested QoS metrics, for generating policies in order to ensure the SLA enforcement and for the placement execution. The reverse translation is needed during SLA templates generation in order to generate templates expressed in SLA metrics based on the available and supported low level metrics. Moreover, the SLA Enforcer is responsible for tuning the Monitoring and Analysis components with appropriate parameters needed for the SLA enforcement. These include the metrics that should be monitored and the threat thresholds. Also, it provides to the Container Management component placement requirements (how many replicas and in which locations) in low-level terms.

SLA Violation Handling. SLA violation is handled according to policies that are sent to Monitoring. When an SLA violation occurs, a notification is sent to the Accounting and Billing in order for the provider to be charged with the agreed penalty. Moreover, the SLA Enforcer stores appropriate information in the Global View so as to be used for preventing future SLA violations.

Proactive SLA Violation Detection. The SLA Enforcer is responsible for detecting proactively possible SLA violations. The SLA Enforcer calculates for each metric the threat threshold, which is more restrictive than the one signed and appropriate policies, and sends them to the Analysis component. The Analysis component receives monitoring information for this metric and calculates trends and patterns. The SLA Enforcer is notified by Analysis when forecasts that a metric will reach the given threshold. Then, the SLA Enforcer decides on the actions that are needed to prevent the imminent SLA violation and it reconfigures the system with new appropriate policies and monitoring parameters. Examples of corrective action are replica creation, replica movement, or request redirection to a cluster which is less loaded.

The proactive SLA violation detection is based on resource usage experiences and historical data and uses the case base reasoning (CBR). CBR is the process of solving problems based on past experience. In the knowledge database used in Global View the conditions are stored under which a violation is going to be realized, and the preventative actions and solutions that should be performed for avoiding the violations.

Analysis algorithms for proactive SLA violation detection regarding forecasting and normality are implemented based on Map/Reduce processes.

Policies Mechanism. The SLA Enforcement is based on policies that are created runtime according to the SLOs and the monitoring data. The policies concern either monitoring or forecasting metrics. Some policies examples are depicted in Fig. 5.

topic	metric	operation	predicate	threshold	aggregationMethod	filterUnit	period	rule-type
throughput-topic	transaction-throughput	PUT	<	5		ntua,niki,photos		violation
throughput-topic	transaction-throughput	PUT	<	7		ntua,niki,photos		threat-violation
latency-topic	transaction-latency	GET	>	500		ntua,niki,photos		violation
latency-topic	transaction-latency	GET	>	400		ntua,niki,photos		threat-violation
average-throughput-topic	transaction-throughput	PUT	<	5	avg	ntua,niki,photos	500	violation
sla-per-request	transaction-throughput	PUT	<	5		ntua,niki		measurement

Fig. 5. Monitoring policies examples.

Monitoring. The main responsibility of the Monitoring Component is the collection, propagation and delivery of all events generated in the system to their respective recipients. To this end, the component employs an asynchronous message delivery mechanism, and on top of it, a simple distributed rules engine to decide where each event should be transmitted and whether it should be aggregated with other events first. The system distinguishes between a number of different aggregation levels. More specifically a rule can be defined at a per node, per cluster, per datacenter or per cloud level combined with a time frame. These levels have different granularities.

The monitoring component's interactions fall into two broad categories:

Producers of Events in VISION Cloud include the Object Service, CCS, Resource Map, SRE (Storlet Runtime Environment), VM-storlets and low-level metric gathering probes. These services generate events upon user actions or at scheduled intervals. The events are passed to Monitoring which performs various aggregation operations and are finally passed on to consumers. A library provided by Monitoring is used (in python and java) for integrating with the producers.

Consumers of Events. Most management operating layer services, for example CTO, SLA Management, Accounting/Billing, Analysis and Analytics service, depend integrally on the events dispatched by Monitoring. The events can be consumed through a provided library.

The main components are the following:

Vismo-Core. This is the main monitoring instance. There is a unique instance running on each node and its main purpose is to coordinate with the other modules. In the most basic terms, it acts as a conductor of events and as such, can be seen as the backbone of the system, receiving events from the event producers and distributing them to the event consumers. Moreover, it is responsible for collecting locally produced events, performing partial (node-level) aggregation and pushing the events to consumers.

Vismo-Dispatch. The sole purpose of this library is to connect a producer to the locally running monitoring instance. In doing so, events generated in the producer are passed in instantly to the monitoring process. Under the hood the open source zmq library is used, in a pull/push fashion.

Vismo-Notify. The library is used by the various consumers to declare interest in one or more group of events, called topics. Upon registration, the library is responsible for notifying the client of the arrival of new events. The notification happens in an asynchronous fashion to the main client program, in another thread. Here, also, the zmq library is employed, using a PUBSUB mechanism.

Rule System. A basic rule system is used to evaluate every event received and trigger different processing actions, such as partial aggregation or immediate dispatching according to rules.

Aggregator. This module is used to generate new events which are the result of an aggregation method upon a list of raw events. Typically, the aggregation happens over events of the same type. Another option is to collect a number of raw events and group them by a given property field.

Rule Synchronizer. This module is responsible for synchronizing all the instances of the rules engine to contain the same rules.

Vismo-Probes. These constitute various low level probes that are external to the main instance and collect data about CPU and memory usage, network load, etc., per node.

CDMI-Queues Service. This service implements and extends the Notification queues as proposed in the CDMI specification. CDMI specifies a means to define and implement notification functionality that is based on queues.

Rules Propagation Mechanism. In order to allow for new rules to be inserted in the system a new mechanism has been developed that allows for rules to be added, updated or deleted at runtime. Moreover, the mechanism guarantees that the rules will be eventually synchronised across all the instances of the distributed rules engine of the monitoring system. The propagation mechanism works as follows: all the nodes at the cluster level form a multicast domain. A simple election mechanism is used to elect a cluster-head, a datacenter-head and a cloud-head. Once a node receives a request for a rule, it propagates the change to all other nodes in the cluster through a multicast message. Each node is responsible to send an acknowledgement to the cluster-head that the change has been received.. Once the cluster is updated the cluster-head is responsible to contact the datacenter head which in turn informs all the cluster-heads.

Implementation and Design Decisions. Our system is based on widely used protocols. For the services we use REST services. Data exchanges are done with JSON format. The database that we use is Cassandra a distributed database. The catalogs are using GPFS. The SLA schema follows the WS-Agreement. Implementation language is Java, Javascript and Python. For the policies the CDMI protocol is used and for federation OVF and OCCI interface. The technical details of the SLA Management component are abstract for other components, so they are unaware of them: RESTful web services are provided to other components in order to use the SLA Management functionality.

The SLA Management component at VISION Cloud is installed in all the nodes, but only one instance runs per cluster. This decision was taken for failover and performance reasons. Some of the SLA Management tasks are handled in different clusters. Nevertheless, synchronization and other matters should be taken into account. An alternative solution is installing the SLA Management component in one cluster and offering a centralized SLA Management. This solves any synchronization issues that may raise, but the performance is severely impacted. The source code of SLA Management was packaged in a WAR file and was deployed on a Tomcat web container, which was tested in the VISION Cloud testbed.

4 Evaluation

In this section we present an experimental evaluation of our system. The tests were executed on a machine with an AMD Phenom II x4 965 Processor running Scientific Linux 6.1. An event creation client created events with configurable event size and rate. The client used the producer library of the mechanism, which added a timestamp to every event. Each message was then propagated to the monitoring mechanism, which after processing the event forwarded to a consumer which appended a timestamp to the event. Using the two timestamps the latency and throughput were calculated in varying scenarios of event rate generation, event size and rules to be executed. The testbed examined consists of 9 machines, organized into 3 clusters.

In the first set of experiments the rate at which events were generated was kept constant at **1000 events per second** while the size of the events was gradually made larger, starting from 512 bytes up to 10240 bytes. For each specific size a set of 5000 events were generated and the mean throughput and latency were calculated. Moreover the memory used by the mechanism was measured in each run. The results can be seen in following, while a statistical analysis of each graph can be found in Fig. 6.

	1000events/sec			3000events/sec			variable events rate	
	latency	throughput	memory consumption	latency	throughput	memory consumption	latency	memory consumption
MIN	0,0281	983,3073	6,1131	0,0343	2375,4264	3,9766	0,0264	3,3901
MAX	0,0390	1001,3353	134,2253	1,6094	2998,7618	189,1281	0,6211	189,7027
AVERAGE	0,0329	992,5467	50,3893	0,6280	2841,0245	96,5134	0,1990	69,5728
STDEV	0,0029	4,7669	31,8081	0,5565	150,4115	62,3577	0,1877	46,2812
Confidence (95%)	0,0009	1,4961	9,9828	0,1746	47,2059	19,5707	0,0213	5,2459

Fig. 6. Statistical analysis of experiments.

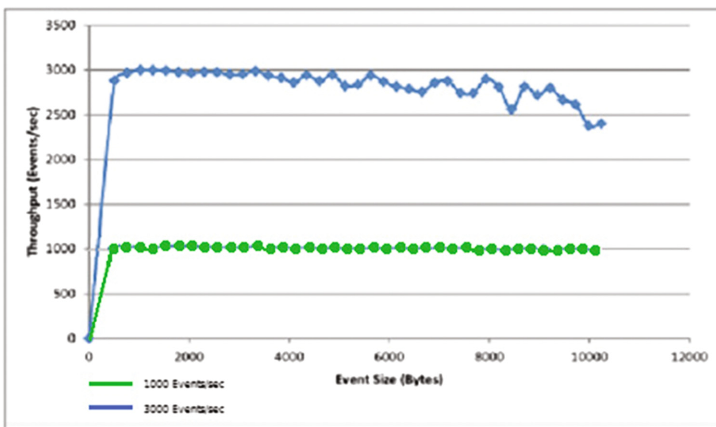

Fig. 7. Throughput with variable size of events 1000 Events/sec rate and 3000 Events/sec rate.

As the size of the events is made larger so does the latency increase, reaching approximately 0,04 s at an event size of 10240. This is considered to be adequate considering that event sizes should not in general be this large. As it is evident the mechanism easily maintains a constant throughput of 1000 events per second. Memory consumption is also affected by the size of the events reaching a maximum of 134 MBs.

In the next set of experiments and in order to stress the system we executed the same experiment with an event rate of **3000 events/sec**. As expected we see a negative impact on the measured latency and throughput (Figs. 7, 8 and 9).

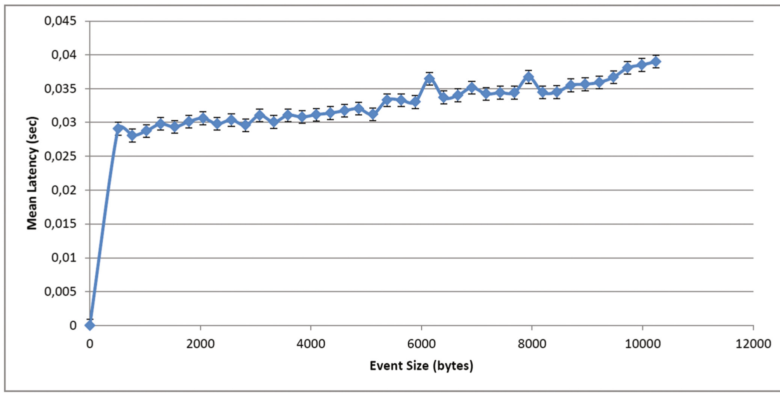

Fig. 8. Latency with variable size of events at 1000 Events/sec rate.

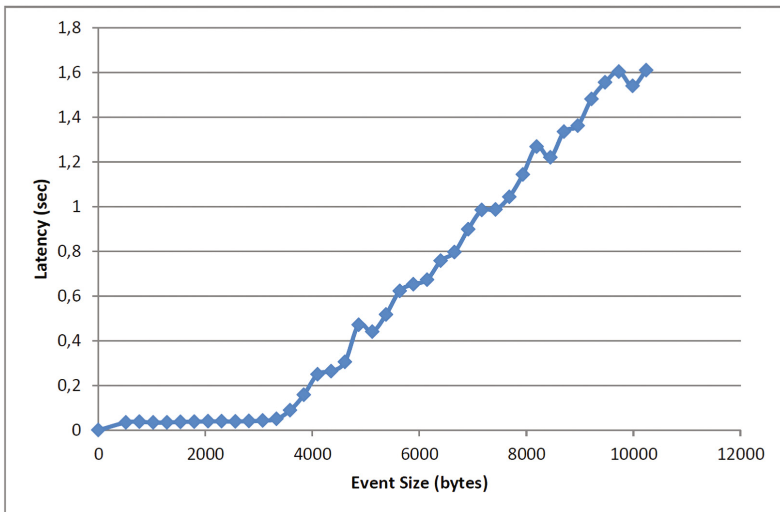

Fig. 9. Latency with variable size of events at 3000 Events/sec rate.

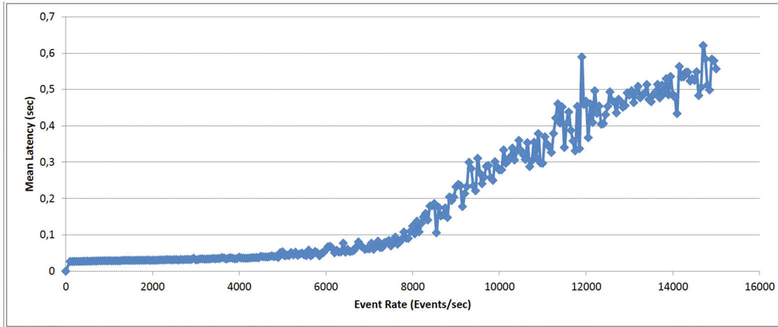

Fig. 10. Latency with variable rate of events.

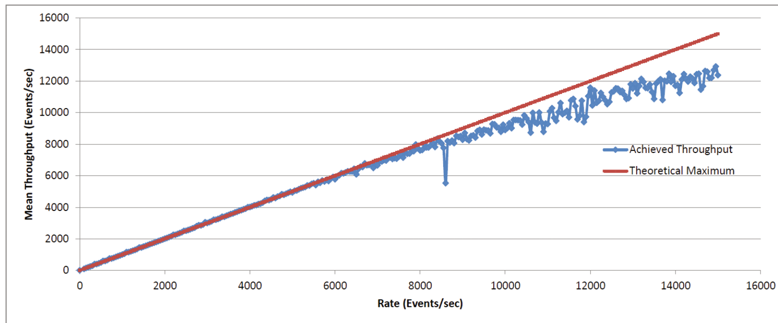

Fig. 11. Comparison of achieved to maximum throughput.

In the third set of experiments the **event size of kept constant** at 1024 bytes while event rates from 100 events/sec to 15000 events/sec were used. For each rate a total of 15000 events were sent and the mean latency and throughput were calculated.

Up to an event rate of 6000 events per second the latency is stable at around 0,04 s. After this point there is a constant increase in its value reaching a maximum of 0,62 s. The throughput of the system is able to easily cope with a generation rate of approximately 6500 events per second. After this point there is a constant increase in the difference between generation and rate and output throughput. Memory consumption is also affected by the rate of the events being similar to the consumption measured during the first two tests and reaching a maximum of 189 MBs (Figs. 10 and 11).

5 Conclusion

In this paper we proposed an automated SLA Management mechanism for content centric storage. It is based on an enriched SLA schema which contains content terms and sections for storlets and federation. SLA Management exploits the chosen content terms and supports services to the customer more efficiently and with reduced cost.

During the SLA enforcement, dynamic rules are created and updated, in order to handle proactively SLA violations. Dynamic SLAs are also supported, as SLA templates are generated according to the current supply, and renegotiation is offered.

References

1. Lamanna, D.D., Skene, J., Emmerich, W.: SLAng: a language for defining service level agreements. In: FTDCS 2003, Washington, DC, USA. IEEE (2003)
2. Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Kakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M.: Web Services Agreement Specification (WS-Agreement)
3. IBM Web Service Level Agreements (WSLA) Project. http://www.research.ibm.com/wsla/
4. Tosic, V., Pagurek, B., Patel, K.: WSOL - a language for the formal specification of classes of service for web services. In: ICWS, pp. 375–381. CSREA Press (2003)
5. Oldham, N., Verma, K., Sheth, A., Hakimpour, F.: Semantic WS-agreement partner selection. In: WWW 2006, pp. 697–706. ACM, New York (2006)
6. Boniface, M., Phillips, S.C., Sanchez-Macian, A., Surridge, M.: Dynamic service provisioning using GRIA SLAs. In: Nitto, E., Ripeanu, M. (eds.) ICSOC 2007. LNCS, vol. 4907, pp. 56–67. Springer, Heidelberg (2009). doi:10.1007/978-3-540-93851-4_7
7. Emeakaroha, V.C., Brandic, I., Maurer, M., Dustdar, S.: Low level metrics to high level SLAs - LoM2HiS framework: bridging the gap between monitored metrics and SLA parameters in cloud environments. In: 2010 HPCS, pp. 48–54 (2010)
8. Brandic, I., Emeakaroha, V.C., Maurer, M., Dustdar, S., Acs, S., Kertesz, A., Kecskemeti, G.: LAYSI: a layered approach for SLA-violation propagation in self-manageable cloud infrastructures. In: 2010 IEEE 34th Annual COMPSACW, pp. 365–370, July 2010
9. Emeakaroha, V.C., Netto, M.A.S., Calheiros, R.N., Brandic, I., Buyya, R., De Rose, C.A.F.: Towards autonomic detection of SLA violations in cloud infrastructures. FGCS (2011)
10. Martino, C.D., Chen, D., Goel, G., Ganesan, R., Kalbarczyk Z., Iyer, R.: Analysis and diagnosis of SLA violations in a production SaaS cloud. In: 2014 IEEE 25th International Symposium on Software Reliability Engineering, Naples, pp. 178–188 (2014)
11. Ghosh, R., Longo, F., Frattini, F., Russo, S., Trivedi, K.S.: Scalable analytics for IaaS cloud availability. IEEE Trans. Cloud Comput. **2**(1), 57–70 (2014)
12. Stanik, A., Koerner, M., Kao, O.: Service-level agreement aggregation for quality of service-aware federated cloud networking. IET Netw. **4**(5), 264–269 (2015)
13. Alrebeish, F., Bahsoon, R.: Implementing design diversity using portfolio thinking to dynamically and adaptively manage the allocation of web services in the cloud. IEEE Trans. Cloud Comput. **3**(3), 318–331 (2015)
14. Zhao, L., Sakr, S., Liu, A.: A framework for consumer-centric SLA management of cloud-hosted databases. IEEE Trans. Serv. Comput. **8**(4), 534–549 (2015)
15. Müller, C., et al.: Comprehensive explanation of SLA violations at runtime. IEEE Trans. Serv. Comput. **7**(2), 168–183 (2014)
16. Morshedlou, H., Meybodi, M.R.: Decreasing impact of SLA violations: a proactive resource allocation approach for cloud computing environments. IEEE Trans. Cloud Comput. **2**(2), 156–167 (2014)
17. Galati, A., Djemame, K., Fletcher, M., Jessop, M., Weeks, M., McAvoy, J.: A WS-agreement based SLA implementation for the CMAC platform. In: Altmann, J., Vanmechelen, K., Rana, O.F. (eds.) GECON 2014. LNCS, vol. 8914, pp. 159–171. Springer, Cham (2014). doi:10.1007/978-3-319-14609-6_11

18. Kolodner, E.K., et al.: A cloud environment for data-intensive storage services. In: IEEE CloudCom 2011, pp. 357–366 (2011)
19. Mavrogeorgi, N., Gogouvitis, S.V., Voulodimos, A., Kyriazis, D., Varvarigou, T.A., Kolodner, E.K.: SLA management in clouds. In: CLOSER 2013, pp. 71–76 (2013)
20. Franke, U., Buschle, M., Österlind, M.: An experiment in SLA decision-making. In: Altmann, J., Vanmechelen, K., Rana, O.F. (eds.) GECON 2013. LNCS, vol. 8193, pp. 256–267. Springer, Cham (2013). doi:10.1007/978-3-319-02414-1_19