Trustworthy Detection and Arbitration of SLA Violations in the Cloud

Christian Schubert, Michael Borkowski(✉), and Stefan Schulte

Distributed Systems Group, TU Wien, Vienna, Austria
{c.schubert,m.borkowski,s.schulte}@infosys.tuwien.ac.at

Abstract. In cloud computing, detecting violations of Service Level Agreements (SLAs) is possible by measuring certain metrics, which can be done by both the provider and the consumer of a service. However, both parties have contradicting interests with regards to these measurements, which makes it difficult to reach consensus about whether SLA violations have occurred.

Within this paper, we present a solution for measuring and arbitrating SLA violations in a way that can be trusted by both parties. Furthermore, we show that this solution is not intrusive to the service and does not incur a significant overhead system load, but nevertheless provides high accuracy in detecting SLA violations.

1 Introduction

Cloud computing offers a significant increase in flexibility and scalability for businesses offering their services to customers [2]. While cloud computing enables features like elasticity and paradigms like Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [18], all of which have proven to be highly effective tools in both industry and research [2], it also features a high level of distribution. This means that software components and services created and operated by various providers need to inter-operate with each other. As software development moves towards adopting Service-Oriented Architectures (SOA) [25], quality and reliability of individual services become important aspects [11], and parallel to agreeing on the service provided and consumed, *Service Level Agreements* (SLAs) are negotiated [30].

SLAs play a major role in cloud computing [8], and find application in grid computing, SOA, or generic Web services [24]. SLAs, negotiated between the provider and the consumer of a service, specify the relationship between those two signing parties regarding functional and non-functional requirements, such as availability, response time or data throughput. A key aim of SLAs is to protect the service consumers, as penalties can be defined for non-compliance with agreed constrains. For instance, the consumer can be awarded with credits by the provider upon detection of SLA violations [4].

SLA violations can be detected by real-time monitoring of the provided services and their runtime environment [25]. While this allows the consumer to

K. Kritikos et al. (Eds.): ESOCC 2018, LNCS 11116, pp. 90–104, 2018.
https://doi.org/10.1007/978-3-319-99819-0_7

detect SLA violations and benefit from penalty payments, the provider can use monitoring to avoid resource over-provisioning and unnecessary cost [6,15].

However, a fundamental problem with SLA monitoring is that both signing parties have contradicting interests with regards to the outcome. For instance, the provider might want to conceal an increase in response time, while the consumer is interested in revealing this violation to benefit from a penalty charge. Therefore, the *trust* of both parties in the measurements, and the arbitration whether an SLA violation has occurred, must be preserved to reach consensus.

In this paper, we provide a solution to ensure trustworthy measurement and arbitration of metrics to detect SLA violations in the cloud. Our solution does not significantly impact the service performance for either party. It allows to monitor detailed application-level details instead of only generic system-level metrics. We propose a hybrid approach, combining dedicated measurement software (agents) with Aspect-Oriented Programming (AOP). Furthermore, we propose a Trusted Third Party (TTP) component which uses Complex Event Processing (CEP) to efficiently handle high-volume data and automatically transforms SLA requirements into CEP expressions to utilize the high scalability of CEP engines. The presented approach does not require modifications to underlying software or protocols.

In summary, this paper provides the following contributions:

- We present a solution for using a TTP component for reliable and transparent SLA arbitration. Our solution is a hybrid approach, using AOP as well as agent-based monitoring, and is transparent for the service as well as its consumer.
- We propose the usage of CEP, together with an automated SLA-to-CEP mapping, to detect SLA violations in an effective and scalable manner.
- We provide a reference implementation of our solution, and valuate its accuracy and performance impact in a testbed.

The remainder of this paper is structured as follows: In Sect. 2, we provide background later used in Sect. 3 to describe our approach. We then evaluate the approach and its implementation in Sect. 4. In Sect. 5, we discuss related work. Finally, we conclude and give an overview of future work in Sect. 6.

2 Background

SLAs, i.e., contracts between service providers and consumers [19], play a major role in cloud computing [8], and are also found in the field of grid computing [19]. Conceptually, SLAs are applicable to any kind of service provided from one stakeholder to another, where not only the functionality, but also non-functional agreements must be negotiated. While we describe our approach in the context of cloud services, the work presented in this paper is not limited to any specific service paradigm.

The design of SLAs is usually tightly coupled with service selection, since it is in the interest of the client to select the provider with the most favorable

SLAs while maintaining a moderate price, and at the same time, it is in the interest of the provider to avoid defining overly strict SLAs. Several approaches for SLA negotiation have been presented in literature [7,30]. Therefore, we do not consider this negotiation phase. Instead, we assume that there is consensus about the SLAs in effect between the two parties.

Furthermore, there is a gap between non-functional business requirements, which are often high-level specifications, possibly provided by non-technical staff, and low-level metrics, which can be directly monitored by software [10]. There exist several approaches in present literature [28] for performing a mapping between high-level specifications and low-level metrics. Again, we assume that this mapping has already been performed, and that the automated measurement of low-level metrics is sufficient to detect SLA violations.

There is a variety of concrete SLA metrics observed by various solutions. We refer to the OASIS Open Standard for Web Services Quality Factors (WS-Quality-Factors) [23], defining quality levels for Web services, e.g., business value quality, service level measurement quality, manageability quality or security quality. The service level measurement quality comprises quantitative, dynamically changing attributes which describe the Quality of Service (QoS) [20]. Consequently, these attributes are highly suitable for real-time SLA monitoring. These metrics are generally in line with metrics found in other literature [1,12]. Both metrics experienced by the client as well as metrics observable on the server are of interest. In detail, we monitor the response time, throughput, availability, successability, CPU and memory usage of cloud services, since these metrics are applied in many different cloud solutions in research and practice [17,21].

3 Trustworthy SLA Monitoring

As discussed in the previous sections, we aim to monitor and arbitrate SLAs in a way that does not require the two signing parties (provider and consumer of a service) to trust each other. In this section, we discuss the overall architecture as well as the individual components of our approach.

3.1 Architecture Overview

We present the architecture of our solution in Fig. 1. The *service provider* is responsible for providing a certain *service* to the *service consumer* running a *client* to access the service. To simplify the figure, we depict only the communication path between the service provider and one consumer. However, the service is not restricted to only one consumer. We use a common message broker to exchange information between all components in a unified and scalable way. We employ *AOP advices* on both sides, i.e., pieces of code injected into the application, responsible for detecting service requests and responses and transparently monitoring application-level metrics. Furthermore, an *agent* is used on the provider side to monitor system-level metrics. Finally, the *TTP* component is provided by an entity not associated with the signing parties, i.e., a *neutral*

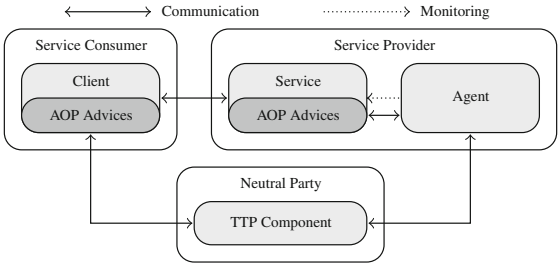

Fig. 1. Architecture overview of the proposed approach

party having no stake in the service and its SLAs. We do not define the process of determining such an entity, but assume that there is consensus between the two signing parties about the usage of a neutral party as a mediator.

Therefore, our solution consists of three main components, in addition to the pre-existing service and client: The agent, the AOP advices, and the TTP component. We describe these components in the following sections.

3.2 Agent

The agent is a stand-alone application hosted by the provider, and is independent of cloud services and other applications. It is responsible for the monitoring of metrics which are either impossible or not feasible to measure from within AOP advices, such as CPU or memory utilization during the execution. We also allow to extend the agent in a modular way by so-called *probes*. Probes act like additional monitoring services, running independently of the agent, but reporting to it. Such probes can be used to measure values from proprietary data sources, or to use other software already deployed. The interaction between the monitored services, the probes, the agent and the TTP is shown in Fig. 2.

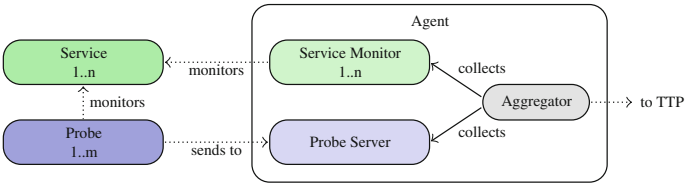

Fig. 2. Architecture of the agent component

The Agent uses Java bindings of the Sigar library[1] to retrieve CPU and memory metrics. Internally, it manages a list of Process Identifiers (PIDs) and

[1] Sigar is a software library to access native operation system and hardware activity information; cf. https://github.com/hyperic/sigar.

monitors their resource usage using Sigar. Furthermore, the agent keeps track of the tree of sub-processes possibly started by the monitored process. In contrast to other work [16], where a list is passed between processes and their children, we obtain the process tree by traversing the system process list, since this does not require changes in application code. Since our agent is aware of the SLA, it only monitors applications requiring the observation of CPU or memory usage, which reduces unnecessary overhead.

Note that it is not the agent's responsibility to interpret data or arbitrate SLA violations. It merely forwards the measurements to the TTP using the message broker. The messages consist of a timestamp, an identifier for the agent and the monitored application, a metric descriptor, and the measured value itself, and is signed using an SHA-256 Keyed-Hash Message Authentication Code (HMAC).

In order to prevent permanent network load and reduce the relative amount of overhead, we consolidate measurements into a queue. At an interval of 5 s, the agent sends the contents of the queue to the message broker. Furthermore, we do not send messages where the measured values diverge less than a certain level from the last transferred value. This is especially useful for processes having long periods of zero or near-zero CPU usage and reduces network traffic as well as computational load of both the agent and the TTP. In our experiments, we have found a threshold of 1% to be sufficient to avoid most of the unnecessary network load. This filtering and aggregation of messages is done in the *Aggregator* sub-component depicted in Fig. 2.

3.3 AOP Advices

In addition to the agent component, we use AOP advices on both the service and the client. AOP advices consist of code injected (*weaved*) into an application, which is executed at well-defined points in the code (*pointcuts*) and allows to transparently monitor software without modifying its source code, while still gaining measurements which would not be possible by the means of agent-based monitoring alone. As shown in Fig. 3, the AOP advices use pointcuts around the request procedures on both sides. Whenever a client is about to send a request to the service, the pointcut triggers the advice, which records the timestamp and request. Similarly, at the service side, whenever a request is received and the handling method is about to be called, the pointcut triggers and the service advice records the timestamp. After the execution of the service, this procedure is repeated for the response in reversed order. The response also contains information about the success or failure of the service, which is recorded by the advices. Finally, all advices independently report their measurements to the TTP component, which then matches the reported measurements, checks them for reasonability and decides whether an SLA violation has occurred.

The AOP advices only differ slightly for the provider and consumer sides. Advices on the consumer side must communicate with the message broker directly, while advices on the provider side are executed on the same machine as the agent, and can therefore report to this agent using local communication.

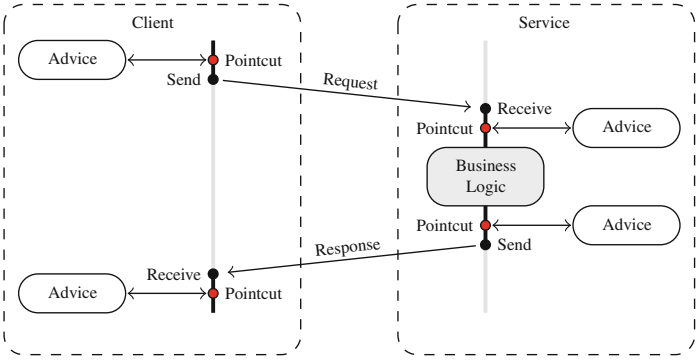

Fig. 3. Operation of AOP advices within Service and Client

3.4 TTP Component

The TTP component is the main unit responsible for detection and arbitration of SLA violations, and is hosted by a neutral party. Architecturally, the TTP component consists of the components shown in Fig. 4. Messages from agents and clients are received from the message broker using the *message receiver* component. They are fed into the *CEP engine* as events. In our implementation, we use the Esper[2] engine, which is used to process these events in an efficient and scalable way. For large-scale systems, this allows for easier outsourcing of this workload onto a distributed CEP system on its own [5]. In our implementation, we use the WSLA standard [12] to define SLAs. The negotiated SLA is read by the *WSLA reader* component. We then map this WSLA instance to CEP expressions in the *SLA-to-CEP mapper*. These CEP expressions are then fed to the *CEP engine*, which, together with the events from the *message receiver*, detects violations.

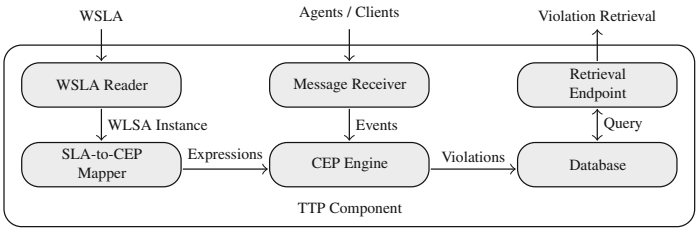

Fig. 4. Architecture of the TTP Component

We distinguish between simple SLA requirements, i.e., a direct mapping of SLA parameters to measured values, and complex SLA requirements, composed

[2] Esper is an open source event processing and correlation solution; cf. http://www.espertech.com/products/esper.php.

Listing 1.1. Extract of an Exemplary SLA Requirement

```
<Obligations>
    <ServiceLevelObjective name="AverageResponseTimeSLO">
        <Obliged>ServiceProvider</Obliged>
        <Validity>
            <Start>
                2017-01-01T14:00:00.000-05:00
            </Start>
            <End>
                2018-01-01T14:00:00.000-05:00
            </End>
        </Validity>
        <Expression>
            <Predicate xsi:type="LessEqual">
                <SLAParameter>
                    AverageResponseTime
                </SLAParameter>
                <Value>2500</Value>
            </Predicate>
        </Expression>
        <QualifiedAction>...</QualifiedAction>
    </ServiceLevelObjective>
    ...
</Obligations>
```

Listing 1.2. EPL Statement Generated by Mapper

```
SELECT AVG(responseTime) AS monitoredvalue,
    'responseTime' AS metrictype,
    'avg<=2500.0' AS requirementdesc, *
    FROM ClientInfoMessage(responseTime >= 0)
    GROUP BY serviceName
    HAVING AVG(responseTime) > 2500.0 AND COUNT(*) >= MIN_QUANTITY
```

from several metrics, possibly applying additional aggregation functions. An example of a simple SLA requirement is "response time lower than 5 s". In contrast, "average response time lower than 5 s" is a complex SLA requirement. The WSLA language defines certain aggregation functions. Our implementation directly translates *Average*, *Median*, *Sum* and *Max* to equivalent expressions in the Event Processing Language (EPL) used by Esper.

As an example, an SLA requirement is shown in Listing 1.1, where the average response time is constrained to 2,500 ms. From this, the mapper creates the EPL expression shown in Listing 1.2. Note that certain sanity checks are already compiled into EPL expressions. For instance, the requirement `responseTime >= 0` filters out negative response durations.

Violations detected by the CEP engine, together with the proof, i.e., the involved events, are then persisted in a *database*, which enables accountability and traceability. Finally, the *retrieval endpoint* can be used to read violations and their information from the database.

4 Evaluation

In our evaluation scenario, a user can request image manipulation (resizing, rotating and flipping) of JPG, PNG and GIF images using a REST interface. The image manipulation service is cloud-based, and the user and provider of the

service have agreed on a WSLA. We use this scenario for our evaluation since image manipulation requires a given computational complexity and is therefore a good placeholder for other possible cloud-based service tasks. For the evaluation experiments, the testbed environment, including the client, is controlled to allow automated repeated experiments with given parameters.

We present the testbed environment together with the image manipulation functionality in this section.

4.1 Testbed Environment

In our testbed, the configuration is provided as a pre-defined WSLA, and the detected violations are output as CSV logs in order to process the experiment results. In our experiment, the neutral party also fulfills the role of the component orchestrating the experiments, i.e., it is responsible for configuring both the provider and the consumer of the service while initiating the experiment.

The image manipulation service provided by the provider is capable of resizing, rotating and flipping JPG, PNG and GIF images using a REST interface. We use this service since this operation represents a given, well-defined computational complexity, and can therefore represent other computational workloads. We deploy the service on an Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instance. EC2 is an IaaS service, which means that we are in control of the operating system and software. Note that since our approach does not require operating system-level operations on the client side, it can also be implemented using an PaaS or SaaS instance for the image service, where the latter requires that AOP advices are supported by the SaaS provider.

We use four image sizes in our experiments: small (640 × 426, 90 kB), medium (1280 × 898, 239 kB), large (1920 × 1280, 692 kB) and huge (4896 × 3264, 2,400 kB). Table 1 shows the employed infrastructure configuration.

Table 1. Testbed environment infrastructure configuration

Instance	OS	CPU (Core Count)	RAM
Provider	Ubuntu 16.04	Intel Xeon E5, 2.4 Ghz (1)	1 GB
Consumer	Windows 10	Intel Core i5, 3.4 GHz (4)	8 GB
TTP	Ubuntu 16.04	Intel Xeon E5, 2.4 Ghz (1)	1 GB

Figure 5 shows an example of an experiment execution by displaying the measured service response time in a series of 500 executions, demonstrating the general functionality of our testbed. We performed the experiment successfully for all metrics, and only show the response time results due to space constraints. The maximum response time is defined as 1,000 ms, and the observed average execution time is slightly higher than 200 ms. We injected deliberate delays of a random duration between 1,000 and 1,100 ms, all 12 of which were detected.

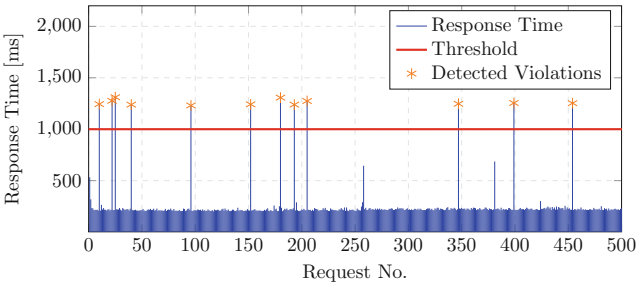

Fig. 5. Example of an experiment execution

For the following experiments, we used 600 experiment runs. Preliminary testing has shown that the system needs a certain amount of time to *settle*, i.e., to create reproducible results. This is most likely due to effects like the Java VM start-up and initial memory allocation. In order to avoid biasing our results with these implementation-dependent artifacts, in our statistical tests, we do not include all observations, but skip a fixed number of observations at the beginning of the experiment (7 for the experiments shown).

4.2 Accuracy: CPU and Memory Usage

We verify the accuracy of our measured CPU and memory usage. We perform t-tests (with a significance level of $\alpha = 0.05$) to verify significant equality of the baseline (see below) and results for the approaches presented in the work at hand. For both tests, the null hypothesis H_0 states equal means of both measurements (baseline and our result), while the alternative hypothesis H_1 states significant deviation between the data sets.

We use the Linux `top` tool in batch mode as a baseline. This tool displays CPU and memory-related information of running processes, which we log and compare to the measurements of our solution. We first measure the CPU usage using an interval of 1,000 ms, since this is the lowest resolution reliably supported by the baseline. We use a batch of 2,000 image resizing requests to create continuous load on our system. Figure 6 shows the overlay of the baseline measurement and the measurement provided by our solution. We also perform a paired t-test for 593 observations, with means of 9.177 and 9.156, and variances of 76.548 and 75.823. The results of the t-test supports our H_0 (p value $0.885 < 1.964$).

From the same datasets, we verify our memory usage measurement. The results are shown in Fig. 7. A paired t-test for 593 observations, with means of 17.162 and 17.161, and variances of 1.484 and 1.485, results again in the support of H_0 (p value $0.693 < 1.964$).

4.3 Successability

For verifying the successability, we inject failures into the service and observe the monitoring outcome. We use a likelihood of 3% of a failure injection, and

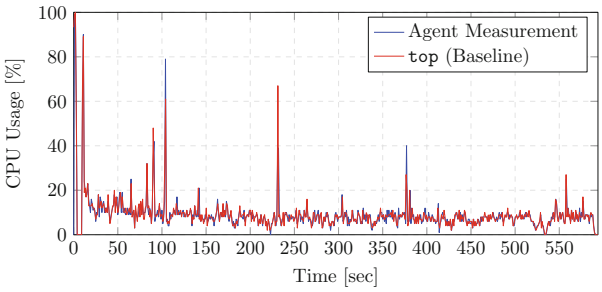

Fig. 6. CPU Usage: Baseline (Red) and Measurement (Blue) (Color figure online)

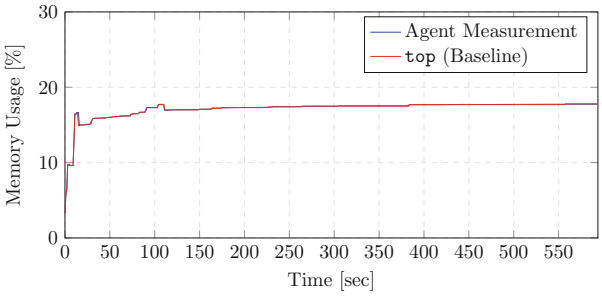

Fig. 7. Memory Usage: Baseline (Red) and Measurement (Blue) (Color figure online)

use 500 successive requests in this experiment. Figure 8 gives an overview of this experiment. The requests, together with their response times (shown on the left axis) are shown in red or green, depending on the outcome. The successability (shown on the right axis) is shown in blue. The effect of each failed request on the successability value can be observed. Also, the successability converges to a value of roughly 97%, corresponding to the 3% failure injection likelihood.

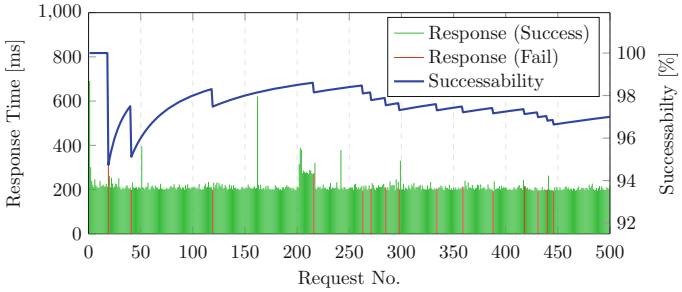

Fig. 8. Successability experiment (Color figure online)

Note that if the first few invocations happen to fail, the measured successability fluctuates subsequently due to the low number of measurements, and possibly incorrect SLA violation are reported. For instance, after around 30 observations, we measure a successability value of 95%, even though the actual baseline value is 97%. We conclude from this experiment that in practice, a lower bound should be set for the total number of observations. We therefore suggest the inclusion of an additional condition to the SLA to restrict the total observation number.

4.4 Performance Impact

Another key aspect, apart from measurement accuracy, is the impact of our monitoring solution on the performance of the service itself. We therefore perform experiments to measure the difference in response time experienced on the client, both with and without our monitoring approach enabled.

When performing the traditional t-tests used in the previous experiments, we encounter the problem that the network between our instances has a significant impact on the response time, seemingly much higher than the monitoring itself. This is indicated by the fact that the executed t-test yield varying results, even for experiment runs with the same configuration, and are therefore inconclusive.

For this experiment, we therefore use a purely local setup with all instances running on a single machine. We use a batch of 1,000 requests per image size and record response times. These results conclusively show that the impact of our monitoring on the response time is negligible. An overview is shown in Table 2. We see that the impact of our solution is well within the standard deviation σ, and never exceeds 3%.

Table 2. Comparison of response times in ms (σ in Brackets)

Workload	With Monitoring	Without Monitoring
Small	27.054 (4.375)	26.524 (3.524)
Medium	53.537 (18.943)	53.383 (17.664)
Big	97.540 (32.987)	95.732 (25.026)
Huge	483.707 (36.481)	482.326 (36.443)

4.5 Maintaining Trustworthiness

Trust into the SLA arbitration approach of both signing parties is crucial in situations where signing parties have contradicting interests. We consider how to maintain trustworthiness on three levels:

Trust in Measurement: The first element of trust is built on the aspect of the accuracy of the measurements. Our solution employs the software architecture described above, which provides reliable and accurate measurements

according to the experiments shown in Sects. 4.1 through 4.4. Ensuring that no measurement is knowingly manipulated on-site can be achieved by using open-source software together with code signing (e.g., by the neutral party hosting the TTP component).

Trust in Communication: Messages exchanged between the components pose potential for deliberate or accidental manipulation. We use ActiveMQ, a well-established open-source broker [27]. In order to further strengthen the trust, we use HMACs to sign our measurement messages, which can be verified by the TTP.

Trust in Arbitration: Finally, both parties must trust the TTP to perform a neutral and fair arbitration. While this trust can be increased again by making the TTP software open-source and by using code signing, in our current implementation, we ultimately rely on the neutrality of the entity hosting the TTP.

5 Related Work

The topic of SLAs has been discussed extensively in existing work. We therefore provide a short overview of the most relevant literature, some of which presents concepts we have adapted in our solution, and conclude by discussing the differences between our work and the two approaches which come closest. We generally classify approaches by their measurement technique, and distinguish between agent-based, middleware-based, and AOP-based monitoring techniques.

A large body of work is found for the mapping of high-level SLA objectives to low-level system metrics. [16] provides an exhaustive overview of this topic, without focusing on the detection of SLA violations, and only takes into account provider-side metrics. Similarly, [22] proposes such a mapping, also taking into account SLA violation detection. The authors evaluate their work in private and public clouds. Again, they do not take into account measurements performed on the client side. Both of these solutions use agent-based monitoring. [13] extends this by using CEP for the detection of SLA violations, a concept we have adopted for our approach.

Instead of using agents, [26] uses a middleware on both sides of the connection. While this enables monitoring of client-side metrics, the authors do not discuss how to use these metrics to verify plausibility, leading to trust increase on both sides, like the checks for reasonable measurements performed by our TTP component. Also, their approach does not provide transparency (non-intrusiveness) to the client and service software. This transparency is provided, however, by works like [29], which provides agent-based monitoring on multiple levels, is extensible and uses a rule language. The authors evaluate their approach using public and private clouds and prove its scalability. [9] also uses agents for measurement, but focuses on predicting SLA violations in the cloud before they happen. [17] uses AOP instead of agents, and also provides transparency.

All of the mentioned approaches, however, do not take into account the trustworthiness of the resulting measurements. Adding a neutral party has been discussed by two approaches which, to the best of our knowledge, come closest to

our solution. [3] suggests a third party similar to our TTP component. However, neither an implementation nor an evaluation is presented. Their solution is using a monitoring agent and is not transparent to the existing code. Furthermore, the solution is limited to the monitoring of communication data, and system resources are not taken into account. [14] also proposes an entity similar to our TTP component, but merely discusses a conceptual framework without providing an implementation or evaluation. Also, this approach is purely agent-based and as such, not transparent to existing code.

6 Conclusion and Future Work

In this paper, we have presented a solution allowing the provider and the consumer of a cloud service to detect violations of defined SLAs, and to allow mutual agreement on the outcome of this detection, without any of the two parties having to trust the other. For this, we have proposed the usage of a neutral third party, which is in charge of the collection of measured values, and the subsequent arbitration of SLA fulfillment.

The neutral third party is hosting the TTP component, which uses a hybrid of agent-based, as well as AOP-based data collection. The TTP component uses CEP to maintain scalability by evaluating the SLA fulfillment using CEP expressions automatically generated from the SLA requirements. Using experiments run in a testbed environment, we have shown that this solution does not only provide accurate measurements, but also does not significantly impact the performance of the service. We have also provided a discussion about various additional aspects of maintaining trust in such a multi-stakeholder scenario.

We currently assume the two parties to trust the neutral third party. In our future work, we plan an extension to our approach using signatures in the verdict of the TTP component, which, coupled with code signing, could remote this requirement and allow for completely trust-less operation. Instead of using a centralized third party, we are currently also observing the possibility of using decentralized consensus, e.g., a blockchain, to increase trust.

Acknowledgment. This work is partially funded by COMET K1, FFG – Austrian Research Promotion Agency, within the Austrian Center for Digital Production.

References

1. Ameller, D., Franch, X.: Service Level Agreement Monitor (SALMon). In: 7th International Conference on Composition-Based Software Systems (ICCBSS), pp. 224–227. IEEE (2008)
2. Armbrust, M., et al.: A view of cloud computing. Commun. ACM **53**(4), 50–58 (2010)
3. Balfagih, Z., Hassan, M.F.B.: Agent based monitoring framework for SOA applications quality. In: International Symposium on Information Technology (ITSim), vol. 3, pp. 1124–1129. IEEE (2010)

4. Baset, S.A.: Cloud SLAs: present and future. ACM SIGOPS Oper. Syst. Rev. **46**(2), 57–66 (2012)
5. Borkowski, M., Fdhila, W., Nardelli, M., Rinderle-Ma, S., Schulte, S.: Event-based failure prediction in distributed business processes. In: Information Systems (2018)
6. Borkowski, M., Hochreiner, C., Schulte, S.: Moderated resource elasticity for stream processing applications. In: Heras, D.B., Bougé, L. (eds.) Euro-Par 2017. LNCS, vol. 10659, pp. 5–16. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75178-8_1
7. Brandic, I., Music, D., Leitner, P., Dustdar, S.: *VieSLAF* framework: enabling adaptive and versatile SLA-management. In: Altmann, J., Buyya, R., Rana, O.F. (eds.) GECON 2009. LNCS, vol. 5745, pp. 60–73. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03864-8_5
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. Future Gener. Comput. Syst. **25**(6), 599–616 (2009)
9. Emeakaroha, V.C., Ferreto, T.C., Netto, M.A.S., Brandic, I., Rose, C.A.F.D.: CASViD: application level monitoring for SLA violation detection in clouds. In: IEEE 36th Annual Computer Software and Applications Conference (COMPSAC), pp. 499–508. IEEE (2012)
10. 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: International Conference on High Performance Computing and Simulation (HPCS), pp. 48–54. IEEE (2010)
11. Islam, S., Lee, K., Fekete, A., Liu, A.: How a consumer can measure elasticity for cloud platforms. In: 3rd ACM/SPEC International Conference on Performance Engineering, pp. 85–96. ACM (2012)
12. Keller, A., Ludwig, H.: The WSLA framework: specifying and monitoring service level agreements for web services. J. Netw. Syst. Manag. **11**(1), 57–81 (2003)
13. Leitner, P., Inzinger, C., Hummer, W., Satzger, B., Dustdar, S.: Application-level performance monitoring of cloud services based on the complex event processing paradigm. In: International Conference on Service-Oriented Computing and Applications (SOCA), pp. 1–8. IEEE (2012)
14. Maarouf, A., Marzouk, A., Haqiq, A.: Towards a trusted third party based on multi-agent systems for automatic control of the quality of service contract in the cloud computing. In: International Conference on Electrical and Information Technologies, pp. 311–315. IEEE (2015)
15. Mao, M., Humphrey, M.: Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In: International Conference for High Performance Computing, Networking, Storage and Analysis. Article number 49. ACM (2011)
16. Mastelic, T., Emeakaroha, V.C., Maurer, M., Brandic, I.: M4Cloud - generic application level monitoring for resource-shared cloud environments. In: International Conference on Cloud Computing and Services Science (CLOSER), pp. 522–532. Springer (2012)
17. Mdhaffar, A., Halima, R.B., Juhnke, E., Jmaiel, M., Freisleben, B.: AOP4CSM: an aspect-oriented programming approach for cloud service monitoring. In: IEEE 11th International Conference on Computer and Information Technology (ICCIT), pp. 363–370. IEEE (2011)
18. Mell, P., Grance, T.: The NIST definition of cloud computing recommendations of the national institute of standards and technology. In: National Institute of Standards and Technology, Information Technology Laboratory 145 (2011)

19. Menascé, D.A., Casalicchio, E.: QoS in grid computing. IEEE Internet Comput. **8**(4), 85–87 (2004)
20. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: Comprehensive QoS monitoring of web services and event-based SLA violation detection. In: 4th International Workshop on Middleware for Service Oriented Computing, pp. 1–6. ACM (2009)
21. Mirobi, G.J., Arockiam, L.: Service level agreement in cloud computing: an overview. In: International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), pp. 753–758 (2015)
22. Moustafa, S., Elgazzar, K., Martin, P., Elsayed, M.: SLAM: SLA monitoring framework for federated cloud services. In: International Conference on Utility and Cloud Computing (UCC), pp. 506–511. IEEE/ACM (2015)
23. OASIS Open. Web Services Quality Factors Version 1.0. Candidate OA- SIS Standard 01 (2012). http://docs.oasis-open.org/wsqm/WS-Quality-Factors/v1.0/WS-Quality-Factors-v1.0.html
24. Repp, N., Eckert, J., Schulte, S., Niemann, M., Berbner, R., Steinmetz, R.: Towards automated monitoring and alignment of service-based workflows. In: IEEE International Conference on Digital Ecosystems and Technologies (DEST), pp. 235–240. IEEE Computer Society, Washington, DC (2008)
25. Rosen, M., Lublinsky, B., Smith, K.T., Balcer, M.J.: Applied SOA: service-oriented architecture and design strategies. Wiley (2012)
26. Al-Shammari, S., Al-Yasiri, A.: MonSLAR: a middleware for monitoring SLA for RESTFUL services in cloud computing. In: IEEE International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA), pp. 46–50. IEEE (2015)
27. Souto, E., Guimarães, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C.: A message-oriented middleware for sensor networks. In: Workshop on Middleware for Pervasive and Ad-Hoc Computing, pp. 127–134. ACM (2004)
28. Theilmann, W., Yahyapour, R., Butler, J.: Multi-level SLA management for service-oriented infrastructures. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 324–335. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-89897-9_28
29. Trihinas, D., Pallis, G., Dikaiakos, M.D.: JCatascopia: monitoring elastically adaptive applications in the cloud. In: 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 226–235. IEEE/ACM (2014)
30. Wu, L., Garg, S.K., Buyya, R., Chen, C., Versteeg, S.: Automated SLA negotiation framework for cloud computing. In: 13th IEEE/ACMInternational Symposium on Cluster, Cloud, and Grid Computing, pp. 235–244. IEEE/ACM, May 2013