Application-Level Performance Monitoring of Cloud Services Based on the Complex Event Processing Paradigm

Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
{lastname}@infosys.tuwien.ac.at

Abstract

Monitoring of applications deployed to Infrastructure-as-a-Service clouds is still an open problem. In this paper, we discuss an approach based on the complex event processing paradigm, which allows application developers to specify and monitor high-level application performance metrics. We use the case of a Web 2.0 sentiment analysis application to illustrate the limitations we currently experience with regard to cloud monitoring, and show how our approach allows for more expressive definitions of monitored metrics. Furthermore, we indicate how the higher-level metrics produced by our approach can be used to increase application elasticity in an existing cloud middleware.

I. Introduction

Current industrial software engineering practice is seeing substantial interest in the novel concept of cloud computing [1] as a virtualization layer, which enables applications to dynamically scale up and down as required by current workload. This model is often referred to as elastic computing [20]. Elastic computing unlocks a plethora of advantages for application providers, including cost savings and greener IT because of reduced energy consumption. All of these advantages come down to preventing over-provisioning of IT resources by closely monitoring demand and acquiring only those resources strictly required for the application to perform on the targeted quality level.

Hence, it is evident that powerful monitoring facilities are at the heart of elastic computing in the cloud. Trivially, monitoring is always important for complex, distributed, enterprise-scale applications, simply to enable business analysts and engineers to reason about the effectiveness and economics of the application. However, in elastic cloud computing, monitoring becomes even more relevant, as it forms the basis, on which all other advantages of the model are built upon – without means to correctly assess the current resource demands of an application, all the dynamicity of the cloud is of little use. In a way, monitoring delivers the knowledge that is required to make scaling decisions with confidence.

Unfortunately, monitoring of applications running in the cloud is currently under-researched, as compared to related areas, e.g., cloud provisioning [11], scheduling [24] or monitoring of the cloud infrastructure itself [10]. Industrial solutions focus on low-level metrics, such as CPU utilization, memory consumption or network bandwidth. We argue that these metrics, while undoubtedly relevant, do not fully capture the actual performance of the application. Instead, applications should be monitored with regard to how well they perform the tasks they were designed for. To give an example, for a Customer Relations Management application provided via the Software-as-a-Service (SaaS) model, a reasonable performance metric is how long it takes for a customer to generate the reports he is interested in, or how many customers can access the application in parallel. Another fundamental advantage of such higher-level metrics is that they can be used as the basis of expressive service level agreements (SLAs). It is clear that these higher-level, application-specific performance measures are somehow connected to the primitives, such as CPU load, but these connections are not easy to derive or express.

In this paper, we introduce a framework for collecting such application-level performance metrics. We disucss our framework based on the earlier presented CloudScale toolkit for building cloud applications [6], and propose a monitoring infrastructure that uses the complex event processing (CEP) paradigm [8]. We discuss how metrics are

defined based on event streams, which types of events can be used, and how these events are correlated. Furthermore, we illustrate how such rich monitoring information can based used to schedule up- and down-scaling of virtualized resources within CloudScale.

The remainder of this paper is structured as follows. We start off with an illustrative scenario in Section II, which will help further motivate our work. Section III contains some essential background information about the CloudScale framework, which forms the frame of the following contributions. Section IV introduces the event-based monitoring framework, which is the main scientific contribution of this paper. We provide an initial numerical evaluation of our monitoring infrastructure in Section V, and, in Section VI, compare our results with related research. Finally, we conclude the paper with a summary and an outlook on future work in Section VII.

II. Illustrative Scenario

To illustrate the ideas of this paper, we use the case of a company providing Web 2.0 sentiment analysis [15] via a SaaS model. In essence, the company allows customers to register with their service, from which point onwards the company will monitor various social networks for positive as well as negative remarks about the customer and its products. Registered customers can query for detailed reports, which are generated on demand and include near real-time data from fast-moving social networks, such as Twitter.

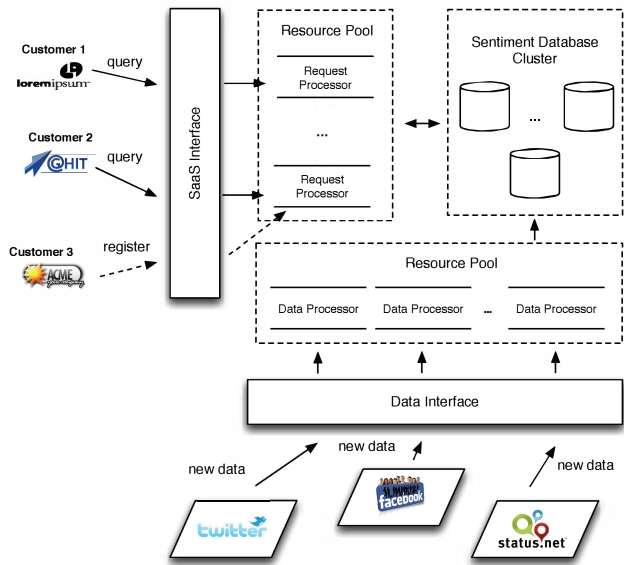


Figure 1: Sentiment Analysis Scenario Overview

To minimize infrastructure costs, and in order to seamlessly scale up and down depending on current load, the sentiment analysis application is hosted entirely using the Amazon Web services[1] (AWS) public cloud. An architectural overview of the application is given in Figure 1. The application is essentially split in two parts. On the one hand, a customer-facing interface abstracts from a dynamic number of cloud hosts handling customer requests (e.g., queries and registration requests). These hosts implement the sentiment analysis business logics, and operate on a cluster of database instances implemented using Amazon's Relational Database Service (RDS). This database is mainly filled by the second part of the application, a data-facing interface, which retrieves real-time data from various social networks. This data is pre-processed by data processing cloud hosts. Pre-processing includes tasks such as stemming or stop-word removal.

Metric Name	Description	Interface
Data-per-second	Avg. number of new data items processed per second	Data Intf.
Time-per-data	Avg. time (in ms) necessary to process a single new data item	Data Intf.
Time-to-sentiment	Avg. time required to get the sentiment on a given keyword	SaaS Intf.
Time-to-report	Avg. time required to generate a report	SaaS Intf.
Time-to-register	Avg. time required to register a new customer	SaaS Intf.
Deployment-costs-per-month	Total costs of the used Amazon resources per month	All

Table I: Example Application-Level Metrics for the Sentiment Analysis Application

In the context of this application, we can think of various meaningful application-level monitoring metrics. Some examples are given in Table I. Clearly, these metrics are not necessarily on the same level of abstraction, and some of the metrics depend on each other, e.g., there is a clear relationship between data-per-second and time-per-data. However, all of these metrics have relevance in the business domain of the application, and, together, they deliver a more accurate of view of the current performance of the application than just looking at, for instance, the average CPU load of the cloud hosts running the application. These metrics allow us to define policies for the scaling behavior of the application, e.g., by defining that the request processor resource pool needs to be increased if the metrics concerning the SaaS interface are too high. The last metric in Table I differs from the others, in the sense that this metric does not really consider the performance of the application, but the costs of delivering this performance. Such metrics are often interesting, because real applications usually cannot be scaled up indefinitely, as there is a cap on how much the provider of the application is realistically able and willing to pay for her infrastructure.

[1] http://aws.amazon.com/

III. Background

While the concepts discussed in this paper are general (i.e., can also be used for other cloud frameworks and applications), our discussion is mostly based on the CloudScale middleware for transparently scaling Java applications [6]. The overall idea of CloudScale is to use the concept of aspect-oriented programming (AOP) to dynamically modify the bytecode of executed applications, and transparently move designated parts of the application (so-called cloud objects) to different virtual resources in the cloud (so-called cloud hosts). These runtime changes are invisible for the application developer. The general procedure, which follows the `Broker` pattern, is sketched in Figure 2.

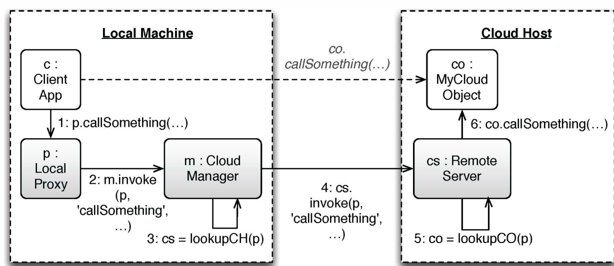


Figure 2: CloudScale Interaction (Adapted from [6])

One of the main tasks (and benefits) of CloudScale is that the framework takes over the management of virtual cloud resources (e.g., virtual machines to run the code on) entirely, as well as the mapping of incoming requests to the (typically replicated) processing instances [7]. To this end, CloudScale requires sophisticated means to track the performance of the application, to decide if more or less resources should be acquired from the cloud. These decisions are governed by scaling policies, user-defined and application-specific rules, which decide, based on monitoring data, whether the current resources assigned to a specific type of cloud object are adequate. Please refer to the original publication for more details [6].

This initial version of CloudScale enabled scaling policies only on very limited, system-level, performance metrics, such as the current CPU load on cloud resources. This makes writing meaningful scaling policies very complicated for application developers, as it is not clear without extensive experimentation what CPU loads are actually optimal for different cloud objects in an application. Instead, application developers would actually like to write scaling policies based on meaningful higher-level metrics, such as the responsiveness of their application for certain time-critical operations. Hence, this paper will discuss how we designed an event-based monitoring infrastructure, which delivers exactly this sort of expressive monitoring data. This data can then be used in scaling policies, but is also relevant as-is. For instance, the monitoring data can also simply be displayed to a human operator in form of a dashboard.

IV. Event-Based Monitoring of Cloud Applications

In the following, we discuss an approach to extend an existing cloud middleware (CloudScale) with event-based monitoring facilities. However, we argue that the presented approach is general, in the sense that the same architecture can also be implemented in other middleware, or even in cloud applications built from scratch (i.e., without any explicit middleware support).

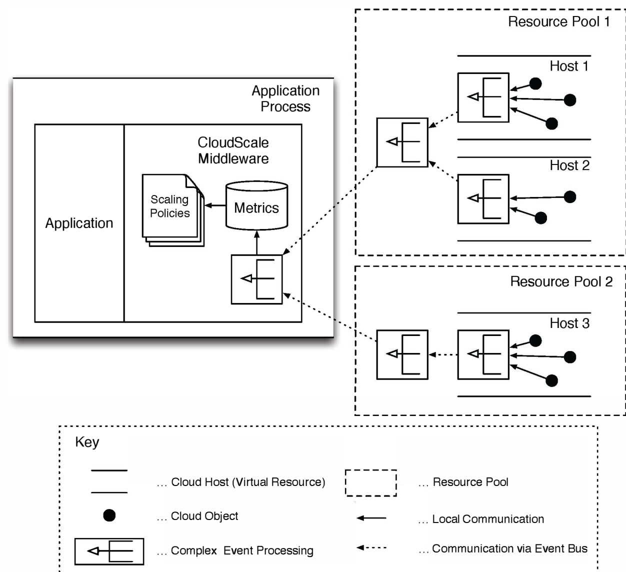


Figure 3: Monitoring Overview

A. Overview

Overall, our monitoring approach is based on the notions of event-based monitoring and CEP. To this end, the idea is that various relevant components in the system emit events, which indicate their current status (e.g., that a new cloud object has been deployed, or that a cloud object has started to execute a given method). We refer to these components as **event emitters**. Important event emitters include cloud objects, cloud hosts or the cloud middleware itself, but, in principle, any Java code executed in the cloud can act as an event emitter by writing into the respective event stream. The events produced by those event emitters are then processed into higher-level knowledge using CEP techniques (a process that we refer to as event correlation

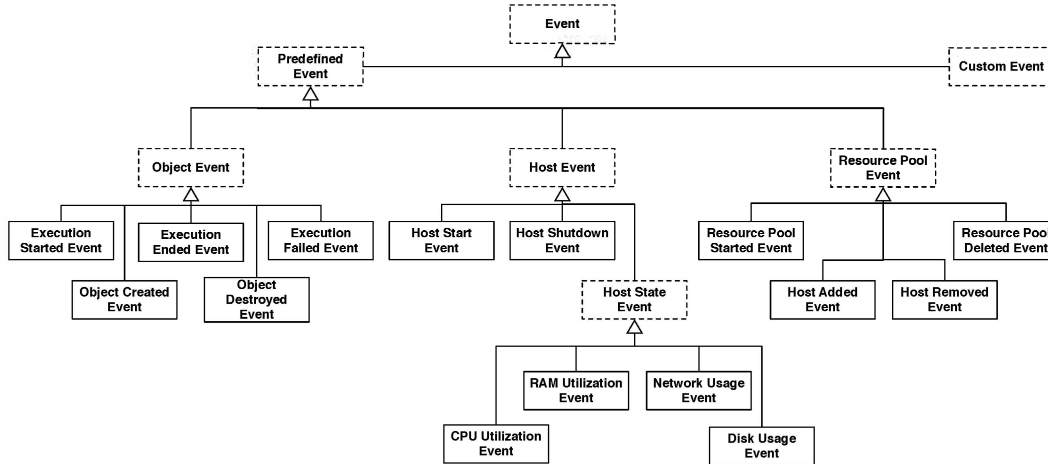


Figure 4: Monitoring Event Hierarchy

in this paper). Finally, monitoring metrics are defined on top of these higher-level complex events. For performance reasons, events are correlated in multiple steps.

Figure 3 sketches the architecture of our proposed monitoring system. Following the CloudScale notion, an application is split into a client application which delegates processing-intense tasks to various virtualized cloud hosts. Logically, these cloud hosts are grouped into resource pools. Each cloud host is executing an arbitrary number of cloud objects. Each component in the architecture can act as event emitter. Locally, events are transported via simple API calls. However, for remote communication (e.g., to send an event from a cloud host back to the application), we use an event bus, e.g., a Java Messaging Service (JMS) implementation or a cloud message queuing service, such as Amazon's Simple Queue Service (SQS). The produced events are correlated on three levels. (1) Firstly, events emitted on a host are correlated locally. This includes events emitted by cloud objects and this cloud host. This step is referred to as **host correlation**. (2) The next correlation step is on resource pool level (**resource pool correlation**), which mainly correlates events generated by host correlation, enriched by additional new events, e.g., events emitted on resource pool level. (3) Finally, the last and most important correlation step is in the application process itself, where all remaining events are used to produce actual metric values, which are then stored persistently into a metrics database. We refer to this final correlation step as **metric correlation**. All these correlation points are technically implemented by different CEP engines. However, in some cases, it may make sense to use one physical engine as implementation of e.g., a host correlation point and a resource pool correlation point at the same time. This is particularly true if different queries are operating on the same event streams, which allows query co-locating [2]. Values in the metrics database can be used in scaling policies, to govern the elasticity behavior of the application (or simply displayed in a dashboard for a human operator). Both, host and host pool correlation, are optional (i.e., in a given application, it is not required to actually use these levels – an application can also just pass on all received events to the next level). Contrary, metric correlation is mandatory, as this step actually produces usable monitoring values from the event stream.

B. Monitoring Event Hierarchy

As indicated in Section IV-A, the basic building block of our monitoring approach are the events generated by event emitters. Hence, Figure 4 depicts the event type hierarchy of all currently available abstract and concrete event types. On the most fundamental level, events can be either predefined or custom. In the current version, our approach considers a set of 15 of the most important runtime events, including lifecycle events for hosts and resource pools, as well as a number of events considering the state of cloud objects. These events are triggered, for instance, when cloud objects start to execute, finish, or fail. This set of predefined events is comparable to the available monitoring events in related systems [3], [12]. All predefined events have an event payload consisting of event-specific additional information. For instance, the `Execution Started Event` carries the unique identifier of the executing cloud object, the name of the method, the parameters of the method, and a generated unique execution identifier as additional parameter. In Figure 4, these additional details are omitted for clarity.

Via the placeholder of custom events, application developers can integrate any application-specific events. This allows developers to extend the monitoring event hierarchy

```
public class MyEventEmitter {
  @EventSink
  MonitoringEventSink eventSink;
  ...
  private void triggerEvent() {
   CustomEvent myEvent = new TweetProcessedEvent(...);
   eventSink.emitEvent(myEvent);
  }
  ...
}
```

Figure 5: Triggering Custom Events

in any way they see fit. For instance, in the sentiment analysis case, a developer may emit a custom `Tweet Processed Event` whenever the application has successfully imported a tweet from Twitter, with the unmodified as well as the stemmed tweet text as payload.

Practically, our framework allows for custom events to be triggered in two alternative ways. Firstly, and more commonly, the CloudScale framework allows to inject an event sink into cloud objects using the well-known `Dependency Injection` pattern. The listing in Figure 5 exemplifies the procedure. All events need to subclass the abstract Java class `CustomEvent`. Events published over injected event sinks are automatically available for correlation on all levels. However, this approach is not available to applications not executing within a CloudScale environment. However, such applications can still trigger custom events by writing events directly into the event bus, e.g., by looking up the correct JMS event queue and writing to it. Such events are available for metric correlation only.

C. Defining Correlation and Metrics

Emitting the basic events described in Section IV-B is, of course, only half the story. These events still need to be aggregated, processed and, potentially, enriched with external data to generate useful monitoring metrics. To this end, application developers define so-called **correlations** and **metrics**. Correlations are named CEP statements, which are executed in defined correlation points (see Figure 6). Potential correlation points are indicated by the "Complex Event Processing" symbols in Figure 3, e.g., for a specific host, or for a specific resource pool. Correlations essentially filter and aggregate low-level events, so that not every single event emitted anywhere in the system needs to be sent to and evaluated directly in the application process. As a convention, each event of a type not used in any correlation is forwarded without modification. That means that, if no correlations are defined, all events are simply passed on to the next correlation level.

In essence, metrics are extended correlation definitions. Metrics are always executed in the same correlation point

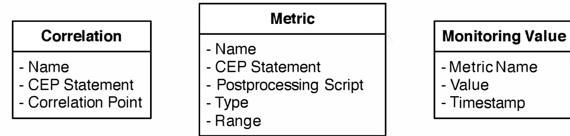


Figure 6: Definition of Correlations and Metrics

(that is, in the final correlation step in the application). Furthermore, the result of the correlation of metric definitions is not forwarded, but used directly as input to metric calculation. Hence, metrics contain, as additional definitions, a script which is to be applied to the final complex events, as well as the type and range of the result. Postprocessing scripts can be trivial (e.g., simply extracting some event properties) or rather complex, querying external data sources (e.g., a pricing service delivering the current per-instance costs of the cloud infrastructure) and enriching the collected data. The purpose of these scripts is to convert the final complex events to actual **monitoring values**. Monitoring values refer to a metric, and consist of the actual value (e.g., an integer) and a timestamp (see also Figure 6). This data is then saved to the metrics database for further usage.

D. Scheduling Based on Monitoring Values

In the context of our work, the main use for monitoring values is to enable expressive scheduling policies, which make sure that the application only holds the virtual resources that are strictly required. Here, we briefly exemplify scheduling policies operating on this monitoring data, to give the reader an idea about possible usage of application-level monitoring in cloud environments.

Scheduling policies take the form of, usually relatively simple, event-condition-action (ECA) rules. The triggering event is generally the reception of a new monitoring value of a given type. The condition usually uses the monitoring value, often to check if it is in a specific target range. Finally, the action is mostly to either increase or decrease the number of hosts in a given resource pool. An illustrative (and simplified) scaling policy is given in the listing in Figure 7.

```
rule 'Add-Data-Processing-Node'
  if time-per-data > 2ms
  then addHost('dataProcPool', 'dataProcHost')
```

Figure 7: Example ECA-Based Scheduling Policy

Evidently, practical problems, such as detecting conflicting rules or preventing oscillation (i.e., continuously triggering up- and down-scaling in short order) need to be

considered, however, these aspects are out of scope for this paper.

E. Case Study Examples

To further illustrate how correlations and metrics can practically be defined, we now present two example metrics based on the sentiment analysis scenario. In these examples, we use the Esper[2] CEP engine for event correlation.

Firstly, let us look at a potential implementation of the metric "Time-Per-Data" (as hourly average), as listed in Table I. We define this metric as the average time between matching `Execution Started Event` and `Execution Finished Events` for the method `DataProcessor.processDataItem(...)` in any given hour. These events are two predefined events, as discussed in Section IV-B. The listing in Figure 8 firstly shows a correlation statement (in Esper Processing Language, EPL, notation), which generates a `ProcessingTimeEvent` for each processed data item (lines 1-9). The correlation is executed in the application, i.e., the metric correlation point is used. Afterwards, the second statement (lines 11-13) defines the actual metric based on this stream of complex events produced by the previous correlation (as the average over a time window of one hour). In this simple example, the postprocessing script of the metric will simply extract the value "TimePerData". The type of this metric is decimal, and the range is $[0; \infty]$.

```
1  // correlation statement (metric corr.)
2  insert into ProcessingTimeEvent
3  select (f.timestamp − s.timestamp) as processingTime,
4    f.executionId as executionId
5  from ExecutionStartedEvent.win:length(10000) s,
6    ExecutionFinishedEvent.win:length(10000) f
7  where s.className = 'DataProcessor' and
8    s.method = 'processDataItem' and
9    s.executionId = f.executionId
10
11 // metric definition
12 select avg(processingTime) as TimePerData
13 from ProcessingTimeEvent.win:time_batch(1 hour)
```

Figure 8: Definition of Time-per-Data Metric

Secondly, we want to implement the "Data-Per-Second" metric. Let us assume that there are two types of data items, Twitter tweets and Facebook status updates. Both are processed by different resource pools. Whenever a data item is successfully processed, the application developer emits custom `Tweet` or `Status Processed Events`, respectively. We follow again a two-step approach. However, this time, we execute a correlation on resource pool level, to aggregate the tweets and status updates for each resource pool (see first statement in Figure 9). The metric definition (second statement in the figure, lines 10-13) can then simply sum up these preliminary results from the two resource pools. The script which defines the metric utilizes the Esper pattern matching facility. The type of the metric is integer, and the range is $[0; \infty]$.

```
1  // correlation statement (resource pool corr.)
2  insert into TweetsPerSecondEvent
3  select count(t) as counter
4  from TweetProcessedEvent.win:time_batch(1 sec) t
5
6  // assume StatusesPerSecondEvent is defined
7  // analogously for status updates
8
9  // metric definition
10 select (t.counter + s.counter) as DataPerSecond
11 from pattern [every (
12  s=StatusesPerSecondEvent and t=TweetsPerSecondEvent
13 )]
```

Figure 9: Definition of Data-per-Second Metric

V. Evaluation

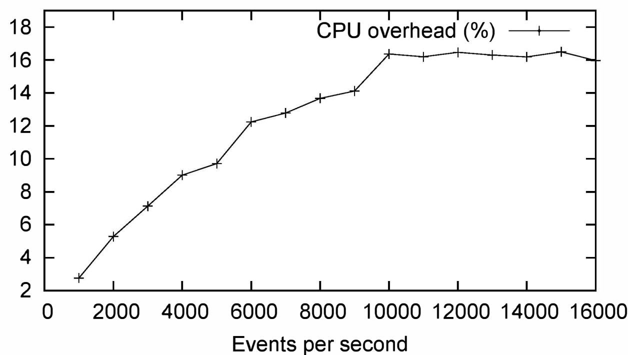
In this section, we discuss a preliminary numerical evaluation of our ideas. We have executed our experiments in a private cloud setting, consisting of a cloud controller and five compute nodes, each running on a dedicated Dell blade server with two Intel Xeon E5620 CPUs (2.4 GHz Quad Cores) and 16 GByte RAM. This private cloud has been set up using the open source Infrastructure-as-a-Service middleware Openstack[3]. The evaluation is conducted using a total of 14 compute instances with the following configurations: one instance hosts an Apache ActiveMQ[4] messaging service (4 VCPU, 8GB RAM), one instance hosts the application process gathering monitoring data (4 VCPU, 8GB RAM), and twelve worker instances emit events (2 VCPU, 4GB RAM).

To demonstrate the feasibility of our approach, the experiment setup resembles a single resource pool where all events are directly consumed by the application process. This represents an edge case with regards to network and CPU overhead on the application process, as there are no aggregation or preprocessing steps performed by host and/or resource pool nodes. Hence, the results presented in Figure 10 do not resemble the overall capacity limits of our approach, but only the limits for this particular resource pool. The presented figures show CPU and network overhead of our approach at the application process instance for different loads, represented by the number of events emitted per second.

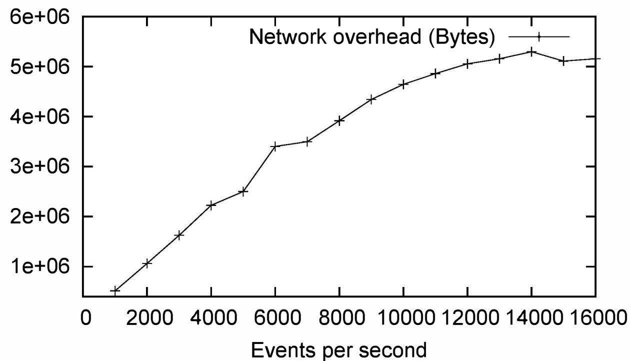
[2] http://esper.codehaus.org/

[3] http://openstack.org/

[4] http://activemq.apache.org/



(a) CPU Overhead



(b) Network Overhead

Figure 10: Experiment CPU and Network Overhead

The events emitted by worker nodes are about 500 bytes in size, and the complex event processing and rule engine modules in the application process are designed to handle roughly 15000 events per second.

As seen in Figure 10a, the processing overhead in our experiment remains reasonably low even at high event rates, reaching around 16% at 10000 processed events per second. The CPU load does not rise significantly for higher event rates as the network connection to the message queue becomes saturated, and the application reaches its designed processing limit, causing an event backlog in the message queue (not shown).

The network overhead of our monitoring framework is shown in Figure 10b. With rising event rates, we see a steady increase of network traffic up to the capacity limit after 14000 events per second. Contrary to CPU load, network load rises until events rates as high as 14000 events per second are reached, due to the data prefetching strategy of the used messaging infrastructure.

Our initial evaluation shows that the proposed monitoring framework is feasible even for high rates of events of considerable size. As mentioned above, the presented results show the behavior of a single resource pool, hence the application can linearly scale using the hierarchical approach discussed in Section IV. Every correlation step described in Section IV-A represents an instance of the discussed experiment, allowing for high event rates within a production system while maintaining reasonable overhead.

VI. Related Research

The emergence of cloud computing poses new challenges to software development, application deployment, and system monitoring. A plethora of research approaches have been published in the previous years. In this section we provide a brief overview of previous work in the related areas of cloud-based application management and CEP-based monitoring.

Event-based monitoring is a paradigm that dates back to the pre-Cloud era. One of the first seminal works in this area is the GEM language [9], an event monitoring language for distributed systems. GEM allows for the specification of primitive and composite event types, and defines trigger rules to enable or disable certain tasks in the system. Analysis of event messages is often applied in safety-critical applications for detection of malicious behavior or intrusion attempts, for instance in the EMERALD [16] environment. Event-based measuring of (mostly low-level) Quality of Service (QoS) metrics has been intensively studied in the context of Grid computing [18].

More recently, event processing techniques have been applied to monitoring [13], [17] and prediction [5], [25] of QoS and Service Level Agreements (SLAs) in service based applications and business processes. Service-oriented computing and cloud computing are said to be in a reciprocal relationship [23], hence, the achievements in automated (SLA) monitoring for (Web) services (e.g., [4]) can partly be applied to cloud computing as well. However, as cloud computing provides a more diverse landscape of services on different conceptual layers, novel monitoring and management techniques need to be devised. Research on application management in cloud environments was so far mostly focused on scalability and deployment of different application architectures (e.g., [14]). The combination of application development and performance monitoring in the Cloud, as discussed in this paper, has received less attention.

Knowledge derived from event monitoring can be utilized to support scheduling decisions, as discussed in this paper. In [22] a cost-optimal scheduling and resource allocation algorithm, based on binary integer programming, is presented. Another important aspect to consider is to determine which parts of an application should be out-

sourced to the cloud. In [21], a cost-based model is utilized to balance the trade-off between cost, time and resource requirements. Besides cost-efficiency, scheduling decisions are often based on fairness criteria to allot resources to multiple applications executing in parallel [19]. In our recent work, we have also discussed SLA-aware client side scheduling strategies for infrastructure clouds [7].

VII. Conclusions

We have discussed an event-based approach for monitoring cloud applications. We have introduced a multi-step, CEP-based event correlation approach, which we argue is scalable even for cloud applications using a large number of virtual resources. We introduced a hierarchy of predefined events, which can be used as basis of metric definitions. Additionally, application developers are free to trigger custom events. Finally, we have shown how the previously published CloudScale framework uses the produced monitoring data to dynamically acquire and release cloud hosts. We presented an initial numerical evaluation, which gives an impression of the performance overhead introduced by our monitoring approach. Our future research with regards to these contributions is two-fold. One the one hand, a larger-scale evaluation of our approach is still required. On the other hand we plan to research the applicability of our approach outside of CloudScale. We argue that our monitoring approach is more general, but still need to substantiate this by applying the approach to other frameworks.

Acknowledgement

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 and from the Austrian Science Fund (FWF) under project reference P23313-N23.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] W. Hummer, P. Leitner, B. Satzger, and S. Dustdar, "Dynamic migration of processing elements for optimized query execution in event-based systems," in *Proceedings of the 2011th Confederated International Conference on On The Move to Meaningful Internet Systems*, ser. OTM'11, 2011, pp. 451–468.

[3] D. Karastoyanova, R. Khalaf, R. Schroth, M. Paluszek, and F. Leymann, "Bpel event model," Report 2006/10, University of Stuttgart, Tech. Rep., 2006.

[4] A. Keller and H. Ludwig, "The wsla framework: Specifying and monitoring service level agreements for web services," *Journal of Network and Systems Management*, vol. 11, pp. 57–81, 2003.

[5] P. Leitner, W. Hummer, and S. Dustdar, "Cost-Based Optimization of Service Compositions," *IEEE Transactions on Services Computing (TSC)*, 2012, to appear.

[6] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "CloudScale - a Novel Middleware for Building Transparently Scaling Cloud Applications," in *ACM Symposium on Applied Computing (SAC)*, 2012.

[7] ——, "Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud," in *5th IEEE International Conference on Cloud Computing*, 2012.

[8] D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Professional, May 2002.

[9] M. Mansouri-Samani and M. Sloman, "GEM: a generalized event monitoring language for distributed systems," *Distributed Systems Engineering*, vol. 4, no. 2, 1997.

[10] T. Mastelic, V. C. Emeakaroha, M. Maurer, and I. Brandic, "M4cloud - generic application level monitoring for resource-shared cloud environments," in *CLOSER*, 2012, pp. 522–532.

[11] X. Meng, C. Isci, J. Kephart, L. Zhang, E. Bouillet, and D. Pendarakis, "Efficient resource provisioning in compute clouds via vm multiplexing," in *7th International Conference on Autonomic Computing*. ACM, 2010, pp. 11–20.

[12] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Advanced Event Processing and Notifications in Service Runtime Environments," in *2nd International Conference on Distributed Event-Based Systems (DEBS'08)*, 2008.

[13] ——, "Comprehensive QoS monitoring of Web services and event-based SLA violation detection," in *4th International Workshop on Middleware for Service Oriented Computing*, 2009, pp. 1–6.

[14] R. Mietzner, T. Unger, and F. Leymann, "Cafe: A generic configurable customizable composite cloud application framework," in *On the Move to Meaningful Internet Systems (OTM)*, 2009.

[15] B. Pang and L. Lee, "Opinion mining and sentiment analysis," *Found. Trends Inf. Retr.*, vol. 2, no. 1-2, pp. 1–135, Jan. 2008.

[16] P. A. Porras and P. G. Neumann, "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances," *National Information Systems Security Conference*, pp. 353–365, 1997.

[17] F. Raimondi, J. Skene, and W. Emmerich, "Efficient online monitoring of web-service slas," in *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[18] R. S. Serafeim Zanikolas, "A taxonomy of grid monitoring systems," *Future Generation Computer Systems*, vol. 21, 2005.

[19] H. Sun, Y. Cao, and W.-J. Hsu, "Fair and efficient online adaptive scheduling for multiple sets of parallel applications," in *17th Int. Conference on Parallel and Distributed Systems (ICPADS)*, 2011.

[20] S. Tai, P. Leitner, and S. Dustdar, "Design by Units - Abstractions for Human and Compute Resources for Elastic Systems," *IEEE Internet Computing*, 2012.

[21] H.-L. Truong and S. Dustdar, "Composable cost estimation and monitoring for computational applications in cloud computing environments," *Procedia Computer Science*, vol. 1, no. 1, 2010.

[22] R. Van den Bossche, K. Vanmechelen, and J. Broeckhove, "Cost-Optimal Scheduling in Hybrid IaaS Clouds for Deadline Constrained Workloads," in *3rd IEEE International Conference on Cloud Computing (CLOUD)*, 2010, pp. 228 –235.

[23] Y. Wei and M. Blake, "Service-oriented computing and cloud computing: Challenges and opportunities," *IEEE Internet Computing*, vol. 14, no. 6, pp. 72 –75, 2010.

[24] M. Xu, L. Cui, H. Wang, and Y. Bi, "A multiple qos constrained scheduling strategy of multiple workflows for cloud computing," in *Parallel and Distributed Processing with Applications, 2009 IEEE International Symposium on*, 2009, pp. 629 –634.

[25] L. Zeng, C. Lingenfelder, H. Lei, and H. Chang, "Event-Driven Quality of Service Prediction," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC'08)*, 2008.