Scalable Monitoring System for Clouds

Andre Brinkmann*, Christoph Fiehe†, Anna Litvina†, Ingo Lück†, Lars Nagel*,
Krishnaprasad Narayanan*, Florian Ostermair‡, Wolfgang Thronicke‡
*Zentrum für Datenverarbeitung, Johannes Gutenberg Universität Mainz, Germany
{brinkman, nagell, narayana}@uni-mainz.de
†MATERNA Information & Communications, Dortmund, Germany
{cfiehe, alitvina, ilueck}@materna.de
‡Atos IT-Solutions and Services GmbH, Paderborn, Germany
{florian.ostermair, wolfgang.thronicke}@atos.net

***Abstract*—Although cloud computing has become an important topic over the last couple of years, the development of cloud-specific monitoring systems has been neglected. This is surprising considering their importance for metering services and, thus, being able to charge customers. In this paper we introduce a monitoring architecture that was developed and is currently implemented in the *EASI-CLOUDS* project.**

The demands on cloud monitoring systems are manifold. Regular checks of the SLAs and the precise billing of the resource usage, for instance, require the collection and converting of infrastructure readings in short intervals. To ensure the scalability of the whole cloud, the monitoring system must scale well without wasting resources. In our approach, the monitoring data is therefore organized in a distributed and easily scalable tree structure and it is based on the Device Management Specification of the OMA and the DMT Admin Specification of the OSGi. Its core component includes the interface, the root of the tree and extension points for subtrees which are implemented and locally managed by the data suppliers themselves. In spite of the variety and the distribution of the data, their access is generic and location-transparent.

Besides simple suppliers of monitoring data, we outline a component that provides the means for storing and preprocessing data. The motivation for this component is that the monitoring system can be adjusted to its subscribers – while it usually is the other way round. In *EASI-CLOUDS*, the so-called Context-Stores aggregate and prepare data for billing and other cloud components.

***Keywords*—Cloud monitoring and management, Scalability, Cloud computing**

I. Introduction

Clouds are complex environments consisting of physical hardware, virtual machines and applications. The hardware is hosted in large data centers that can be operated more efficiently than single computers or small server rooms. In order to increase the hardware utilization, the hardware is usually virtualized. Instead of on physical nodes, applications are run on virtual machines (VMs) so that several VMs share the resources of a single physical machine.

Aside from providing hardware and software services, a cloud provider also monitors key performance indicators (KPI) of the systems and services. In computing centres in general, system monitoring helps to detect hardware and software problems, to improve the resource utilization and to ensure the system's performance and security. In cloud systems monitoring is also essential for offering measured services and for minutely charging customers for their resources and services consumed. Finally, the service level agreements (SLA) between cloud provider and customer are verified.

A monitoring system that works well for small clouds does not necessarily work well for large clouds. An increase in the number of (virtual) servers results in an increase of monitoring data and can lead to an overload of components: network, databases etc. [2]. A scalable monitoring system must therefore eliminate all bottlenecks and avoid the collection and storage of unnecessary data.

This paper introduces a monitoring architecture that is developed in the *EASI-CLOUDS* project. We will describe the architecture's components, their interactions and argue why it is scalable. Moreover, we will outline the reference implementation that is currently realized within the *EASI-CLOUDS* project.

The remainder of the paper is organised as follows. Section II presents the monitoring architecture and its components, section III arguments for its scalability. The related work is discussed in section IV before the paper is concluded with section V giving a summary and an outline of future work.

II. Description of Monitoring Architecture

A state-of-the-art cloud infrastructure like *EASI-CLOUDS* requires a wide range of system information in order to provide secure, scalable and billable on-demand services. The cloud needs to be administered and monitored to improve the scheduling of the VMs with respect to SLA compliance and energy efficiency. The architecture described in this paper consists of three types of components, namely the *Data Suppliers*, the *Data Manager* and the *Data Storage and Preprocessing*.

The monitoring data necessary for the aforementioned tasks has to be gathered from various data suppliers such as the infrastructure, the PaaS software and the applications themselves. For convenience, we will concentrate on infrastructure level monitoring, which is the main supplier of monitoring data. From an architectural point of view, the other data suppliers behave in a similar way. That is, they provide monitoring data to the data collecting unit, which is called *Data Manager*.

The *Data Manager* is the centrepiece of our architecture which organizes all the monitoring data and offers methods for adding and retrieving it. Its interface is the access point for consumers and supplies data from diverse sources in a transparent way. The data is arranged in a tree structure based on data handlers which represent one data supplier each. The architecture is completed by the *Data Storage and Preprocessing* component, which is the only component that preprocesses data and retains items for longer periods. In order to avoid the storage of unnecessary data, this component is only used for special tasks. In *EASI-CLOUDS*, the *Data Storage and Preprocessing* keeps the accounts and stores SLA violations for the billing-as-a-service component. Moreover, it memorizes severe errors and security attacks for later analyses. The *Data Storage and Preprocessing* extends the data manager's API to offer its services.

The monitoring system and the data flow are depicted in Figure 1. Components exchange information either via message brokers or web services. The message brokering protocols are AMQP and XMPP. The REST style is used for the web services.

In the following subsections, we will describe the components of the architecture in more detail.

A. Data Manager

The configuration and monitoring of cloud-based services pose a large number of problems because of different management standards and paradigms. Many specifications and protocols have been introduced in the last decades, but not all of them have proven to be effective and hardly any protocol can claim to be a standard [5]. This is a problem for cloud providers because they cannot support all protocols, and if they choose one particular management protocol, then they will limit their choice of components to the ones supported by the selected protocol.

The two main challenges today are the heterogeneity of the management landscape and the different paradigms for remote management involving proprietary and non-proprietary interfaces. There is an urgent need for a general and easy-to-use management data model which allows the transparent integration of all those techniques and principles. This requires a protocol-independent definition of the management data provided at runtime by services or agents via different standards and protocols. The solution lies in the introduction of an abstraction layer which decouples the consumers from the specific data providers, so that these data can be accessed without depending on the underlying service implementation. The cloud environments require a scalable and flexible management system for handling their monitoring and configuration tasks. This comprises modularity, extensibility, being light-weighted together with a high degree of fail-safety and system availability. We are confronted with changing customer requirements and with new services arising at runtime which have to be monitored and configured depending on their individual management capabilities. This requires an innovative management solution realized in the form of an umbrella environment that aggregates and homogenizes the available monitoring and configuration covering the differences in data addition, retrieval and representation.

1) Management Principle: Almost all management standards are based on specific object models and provide primitive

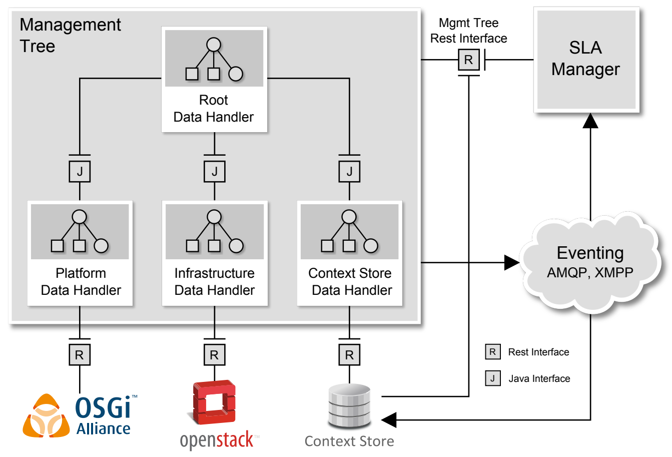


Fig. 1. Tree Composition

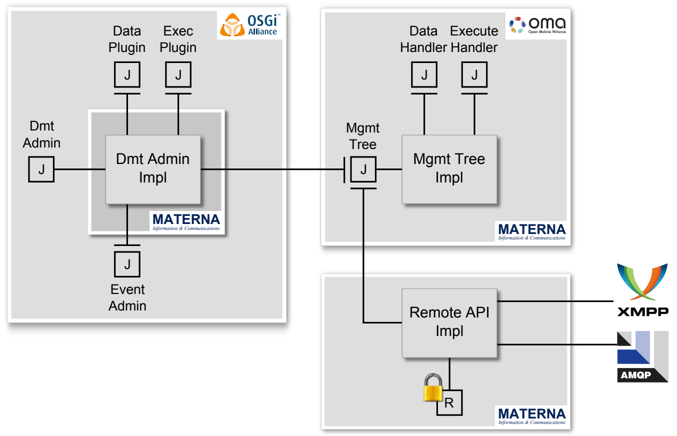
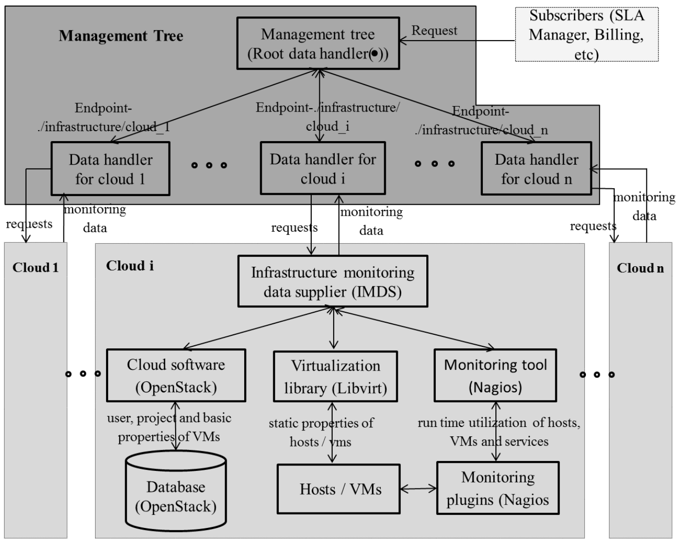


Fig. 2. Architectural Outline

functions. We adapt this approved and well-established management paradigm and extend it by means of a hierarchical data model combined with modification functions such as addition and removal of instances, event notification, etc. In *EASI-CLOUDS*, we propose a solution which is guided by the *Device Management Specification* of the *Open Mobile Alliance* (OMA) [15] and the *DMT Admin Specification* of the *OSGi Alliance* [17] for fulfilling these requirements.

Our approach comprises a distributed management tree that covers all protocol-specific parameters for data acquisition through specific handler implementations. These are termed *data handlers* and they are realized in the form of an independent software component (Figure 1). It can be dynamically provisioned at runtime based on the requirements, on the monitoring and configuration system. This means that through the management system, only those parameters can be accessed that has data consumers. When specific management data needs no longer to be present, its data handlers can be disposed for maximizing the system's performance.

2) Architectural Outline: The architecture of the management tree is depicted in Figure 2 and it is implemented in a component-oriented fashion. The core module contains the base management tree (also termed as *root data handler*) and it offers a Java-based API. There are standard adapter implemen-



Management Tree
Management tree (Root data handler(•))
Request
Subscribers (SLA Manager, Billing, etc)
Endpoint- ./infrastructure/cloud_1
Endpoint-./infrastructure/cloud_i
Endpoint- ./infrastructure/cloud_n
Data handler for cloud 1
Data handler for cloud i
Data handler for cloud n
requests
monitoring data
Cloud 1
Cloud i
Cloud n
Infrastructure monitoring data supplier (IMDS)
Cloud software (OpenStack)
Virtualization library (Libvirt)
Monitoring tool (Nagios)
user, project and basic properties of VMs
static properties of hosts / vms
run time utilization of hosts, VMs and services
Database (OpenStack)
Hosts / VMs
Monitoring plugins (Nagios

Fig. 3. Representation of Data handler for Infrastructure Level Monitoring

tations available for example, DMT Admin, that can be used for accessing the management tree with in OSGi environments. For remote access, a REST-based skeleton implementation is avalaible which exposes the management tree's pull-based access capabilities. Furthermore, the tree uses the Extensible Messaging and Presence Protocol (XMPP) and Advanced Message Queuing Protocol (AMQP) for remotely notifying its subscribers. It is guaranteed that the registered listeners receive only those events for which they are authorized.

At runtime, the *data handlers* are mounted to their corresponding location using their extension points within the management tree. Each implementer organizes his management-relevant information in a self-defined hierarchical structure. As already stated in [6], [7], [8], we suggest a strict separation of monitoring and configuration data. It means that a *data handler* offers isolated subtrees for representing its monitoring and configuration information. This approach guarantees the separation of concerns and takes into account that there is always a temporal offset between the trigger of a state change and its completion. Thus, the reconfiguration process itself and its actual outcome can be monitored by the runtime management system.

B. Data Supplier

The data suppliers are responsible for collecting monitoring data and providing them to the *Data Manager* via *data handlers*. There is a wide range of data suppliers in cloud environments. In this paper, we concentrate on infrastructure monitoring which monitors the cloud's physical and virtual infrastructure (see Figure 3). The cloud system of our architecture is grouped in several sub-clouds that are managed and monitored separately. Each sub-cloud has its own virtualization and monitoring software as well as its own *data handler*.

Each *data handler* structures its infrastructure monitoring data in a subtree that is integrated into the tree of the *Data Manager*. The root of this tree is termed *root data handler*. It has the URI "." and defines *extension points* for other *data handlers*. For example, the *data handler* identified by the REST skeleton */infrastructure/cloud 1* would be mounted at *./infrastructure/cloud_1*. In order to update its subtree, the *data handler* requests monitoring data from the *Infrastructure Monitoring Data Supplier* at regular intervals. This way current values are always readily available for fast access whenever the data is requested via the *Data Manager*'s API. The *data handler* preprocesses the data in order to remove redundancy and to sort them into the tree. The structure implemented in *EASI-CLOUDS* regards the VMs as children of the users and the resources that are assigned to a VM as the children of this VM.

As shown in Figure 3, the data passed to the *data handler* is first aggregated from various sources by the *Infrastructure Monitoring Data Supplier*. Information about the users and their services is provided by the *Infrastructure as a Service* (*IaaS*) or *Platform as a Service* (*PaaS*) software itself together with information about the virtualized infrastructure. Some of these *IaaS* and *PaaS* packages like *OpenNebula* or *Cloud Foundry* offer built-in monitoring tools, whereas others like *OpenStack* (currently) depend on external tools. Common tools are *Nagios*, *Ganglia* and *collectd*, which use plug-ins to monitor the infrastructure.

1) Implementation: The current implementation of *Data Supplier* in the *EASI-CLOUDS* project is a work in progress and based on the software components named in Figure 3: The *IaaS* software is *OpenStack*, an open source project managed by the *OpenStack Foundation*. When the *EASI-CLOUDS* project started, it did not include a monitoring system[1]. OpenStack uses *libvirt* to communicate with the hypervisor and to manage the VMs' operating systems. *libvirt* supports different hypervisors and eases the access by offering a generic API. Finally, the third software is Nagios, a monitoring software that is widely used in computing centers. It comes with thousands of plug-ins, among others plug-ins for *libvirt* and *OpenStack*.

The *Infrastructure Monitoring Data Supplier* was newly implemented in the *EASI-CLOUDS* project. It gathers monitoring data from *Nagios*, *OpenStack* and *libvirt* and offers them via a REST API that is consumed by the respective data handler of the *Data Manager*. The REST API mimics the monitoring interface of *OpenStack*, which was outlined, but not implemented when the supplier component was developed. The data provided by Nagios is retrieved from plugins, especially so-called *check plugins*, which are deployed on the virtual machines. They monitor services and report the resource utilization of virtual machines at runtime. The plug-ins perform system calls on the (virtual) servers and read out dynamic parameters like the current CPU, memory and I/O utilization of a virtual machine. Whenever a new virtual machine is created in *OpenStack*, it has to be registered with Nagios. For the VM's configuration we use *Puppet*.

OpenStack and *libvirt* provide static parameters. *OpenStack* supplies user and project details and information about the hosts, VMs and images (from its database). Additional information about the virtualized hardware is directly polled from *libvirt* which provides more details about the virtual machines, their assigned resources and operating systems.

[1]In the latest release (April 2013), the component *Ceilometer* implements parts of *OpenStack*'s monitoring API.

C. Data Storage and Preprocessing Component

The task of the *Data Storage and Preprocessing* component is to persist and preprocess selected monitoring data, what is neither supported by the *Data Suppliers* nor by the *Data Manager*. The filter and sort operations applied by the *data handlers* could be regarded as data preprocessing, but only insofar as it is necessary to insert the data items at the correct places. The preprocessing performed by this component, on the other hand, serves the provision of context-enriched data. The idea is to shift some of the logic and storage tasks to the monitoring environment. In this way, the monitoring system gets adjusted to meet the subscribers requirements although it is usually the other way around.

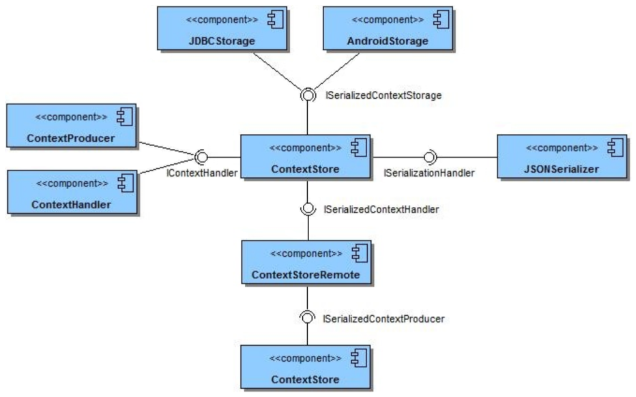
One such example is the billing component that charges the customers for the used services. In the *EASI-CLOUDS* project, it embodies the billing-as-a-service idea and is a plug-in component that can be easily employed in other cloud systems. Since it does not collect the data itself, the *Data Storage and Preprocessing* performs the task of aggregation of the resource usage of users and VMs. Other management tasks that can benefit from stored and context-enriched data are the observance of SLAs and the analysis of system failures.

The results from the *Data Storage and Preprocessing* component are provided to the consumers via the *Data Manager*. Citing the example use case from above, the Billing component requires the aggregated resource usage of VMs. It uses the *Data Manager*'s REST interface for obtaining this information. The tree identifies the *data handler* that is responsible for answering the request and, in this example, it finds the *data handler* of the *Data Storage and Preprocessing* (see Figure 1). The *Data Storage and Preprocessing* component gets the necessary details by registering itself with the tree for the *resource usage events* and the tree notifies it at regular time intervals. It preprocesses and persists the obtained information in a distributed database and publishes them using a REST interface. The data handler then uses this interface for retrieving the aggregated resource usage information of VMs.

1) Implementation: The implementation of the *Data Storage and Preprocessing* is called the context-processing platform, whose main components are instances of the so-called *ContextStore*. The *ContextStore* is ideally suited to combine different data sources and calculate significant contexts. In the area of cloud monitoring, this platform can be used to aggregate arbitrary monitoring events and it provides meaningful input for high level components like billing or SLA management. This is the difference from traditional monitoring approaches, as each cloud service has potentially its own characteristics how it can be monitored and how service delivery aspects have to be combined with low-level monitoring data like cpu-utilization or memory allocation.

The *ContextStore* is based on the event programming paradigm and supports user-defined serialization, transportation and persistence storage mechanism on a plug-in basis. It has the ability to connect to an arbitrary number of other ContextStores distributed across the network. It also offers a high degree of flexibility by using pluggable solution-specific context handlers.

The *ContextStore* persists the data items as *ContextEvents* in a database. This simplifies the design of handlers that do not only need the recent event but also access to previous events to correlate the information. Aside from the payload, the *ContextEvents* have a timestamp, a sender ID and belong to an event type. The *ContextStore* supports local persistence as well as distributed storage. *ContextEvents* may also be propagated between ContextStores. In a typical cloud scenario this is useful as whole resources and, thus, local ContextStores may disappear if no longer needed.



<<component>> JDBCStorage
<<component>> AndroidStorage
ISerializedContextStorage
<<component>> ContextProducer
<<component>> ContextHandler
IContextHandler
<<component>> ContextStore
ISerializationHandler
<<component>> JSONSerializer
ISerializedContextHandler
<<component>> ContextStoreRemote
ISerializedContextProducer
<<component>> ContextStore

Fig. 4. Context Store Architecture

If it is necessary to transport *ContextEvents* across system boundaries to another *ContextStore*, the pluggable ContextStoreRemote component (see Figure 4) provides a REST API or message-based protocol (XMPP) implementation for the data transportation. The components for persistence and serialization are also designed to be pluggable and can be adapted to intended purpose. Currently, support for json/xml serialization and jdbc/android (MySQL, Oracle, sqlite) databases for persisting *ContextEvents* are available. For sending or receiving *ContextEvents* from the *ContextStore*, the interface *IContextHandler* must be implemented. Additionally, a utility package is available which offers pluggable components for bridging OSGi and *ContextEvents*. A recorder and a player module for *ContextEvents* help to test the system. For development of ContextStore clients a REST proxy is available that offers the Java-CS-API directly and hides all the transport specifics.

III. Scalability of the Architecture

This sections analyzes the scalability of our monitoring system. Such a system is scalable if an increase in the number of data sources and requests does not result in the neglectance or loss of data or in an increase in the response time.

Following this definition, the monitoring system has to avoid bottlenecks by automatically adjusting its resources to the cloud's size and the amount of monitoring data. In the following sections, we will explain how the different components of the system achieve this.

The communication between these components is based on REST interfaces and message brokering. Part of the reason why these techniques are used in *EASI-CLOUDS* is their excellent scalability ([9], [21], [16], [4]). The efficiency of REST interfaces can be further improved by using caching, filtering, ETags and the 100 status of HTTP.

A. *Scalability of the Management Tree*

The management tree implementation provides a location-transparent access to the available management data. These data are not stored within the component itself; it rather offers a homogeneous and consistent interface for data retrieval and manipulation.

To ensure scalability within large distributed monitoring environments, the management tree itself is distributed across the network. A management tree implementation can therefore be placed close to those components which require monitoring and configuration. This approach reduces the data that has to be transmitted over the network to a bare minimum. When the monitoring environment grows, e.g. when new components are added or when disjoint monitoring environments are merged, the management tree can be rearranged or extended. In order to maintain location-transparent access of the management data, these management trees can be nested. They can be added to an existing management tree in the form of data handler implementations forwarding calls to the remote data provider.

The data handler approach has several advantages: On the one hand, it allows a transparent nesting of disjoint management tree implementations. On the other hand, it allows the extension of the monitoring environment by means of new data handler implementations. Therefore, new components which require monitoring and configuration can be made visible and accessible within the monitoring domain. The management system is represented in the form of a distributed tree consisting of a variety of different data handler implementations. Within a cloud federation, a cloud provider can authorize other federation members to mount one or more of his subtrees on their own. Thus, each management tree offers an entry point for conducting monitoring and configuration tasks. The monitoring and configuration infrastructure can be adapted at runtime by mounting and unmounting the corresponding data handler implementations, each equipped with specific management capabilities. The management system itself can be customized in order to fulfill the requirements imposed by the surrounding environments concerning SLA monitoring or component deployment.

The separation of data access and data storage ensures that the management tree scales well in high-dynamic environments. It is not responsible for data persistence, because in most cases only a subset of the accruing low-level monitoring data must be stored for future tasks. But it is obvious that there is a need for data pre-processing and aggregation, in order to minimize the volume that has to be actually persisted.

B. *Scalability of Infrastrucure Monitoring Component*

As described in Section II-B1, the data of the *Infrastructure Monitoring Data Supplier* is provided by *OpenStack*, *libvirt* and *Nagios*. We assume that all *OpenStack* clouds (including *libvirt*) are not scaled beyond their capabilities so that scaling the cloud actually implies setting up new *OpenStack* instances once the existing instances have reached their limits.

In this section we therefore only consider *Nagios* and the question whether and how *Nagios* has to be extended to monitor an *OpenStack* cloud. As shown in performance tests ([13], [18]), *Nagios* can efficiently monitor a few hundred services per second, while *OpenStack* can host thousands of VMs. In order to scale with the cloud, *Nagios* can either adapt the time interval of the checks or increase its capability. The latter can be achieved by running several *Nagios* instances on the same cloud or by using a distributed monitoring solution. *Nagios* offers several such solutions, e.g., the load balancers *Mod_Gearman*, *Nagios Fusion* and *Distributed Nagios eXecutor (DNX)* [14], whose task it is to distribute the workload among worker nodes. The system can scale by adding and removing worker nodes.

C. *Scalability of Data Storage and Preprocessing*

As described in Section II-C, the component for *Data Storage and Preprocessing* is realized as a context-processing platform. This platform consists of *ContextStores* that persist and exchange data in form of *ContextEvents*.

The scalability of the storage can be guaranteed by employing cloud databases like *Apache Cassandra* or the *Oracle Database*. Due to the vast amount of monitoring data, it is generally advisable to store as little data as possible. The removal of unnecessary data is supported by the *ContextStore* as it provides the tools for filtering and aggregating data as well as a garbage collection.

For the exchange of *ContextEvents* with another *ContextStore* or with a *data handler* integrated into the *Data Manager*'s tree structure, REST APIs and XMPP are used because of their excellent scalability. The respective XMPP servers can be scaled by *cascading* or by *federation* [10].

Besides storage and communication, the actual data processing could become a bottleneck as well. If this is the case, the respective *ContextStore* can be split into two or more *ContextStores* that keep the connection by communicating via REST or XMPP. In order to reduce their communication to a minimum, the preprocessing tasks will be treated as logical units that are only split if it is absolutely necessary.

IV. Related Work

This section provides an overview of the related work. While there are many papers about cloud monitoring systems, only few consider their scalability.

In [11], Alcaraz et al. present a scalable and elastic distributed monitoring system based on a peer-to-peer architecture, which consists of a *data layer*, a *processing layer* and a *distribution layer*. The data sources on the data layer are described in a metadata language and integrated using specific adaptors. The data is retrieved using SQL-like queries. In the processing layer an operator tree is generated for each query and the distribution layer is responsible for executing the operations across the cloud environment. The authors show in experiments that their monitoring system is elastic and scalable and that new VMs and data sources can be dynamically integrated into the cloud infrastructure. However, their solution does not support storing data for longer periods.

Canali et al. [3] argue that cloud monitoring systems do not scale well due to the large amount of data that is collected from the infrastructure. They propose to identify similar VMs in order to reduce the monitoring data that is sent and stored. Clustering is used to group VMs with similar resource usage and only the data of one representative is forwarded. In an experiment with 110 VMs, running either database servers or webservers, they achieve an accuracy of at least 85% while reducing the data by a factor of 20.

The scalable monitoring framework VOtus, introduced by Suhail et al. [12], is an extension of Otus [19]. It monitors *Hadoop* jobs in virtualized clusters and stores the data in a database for analyzing and debugging the application. They achieve scalability by using Apache HBase, a distributed database system.

Wang et al. [20] describe three architectures for improving the scalability of their *Run-Time Correlation Engine* (RTCE), which correlates and analyzes large volumes of log data from enterprise applications. Besides load balancers, they line out an architecture with so-called *Distributed Event Correlation Engines* (ECE) which gather and locally correlate data before forwarding them to a master node. Since the ECE instances are distributed over the network and since each ECE can work independently, the system can scale with the cloud.

General concepts and properties of cloud monitoring systems are explained and discussed by Aceto et al. [1]. They provide an extensive list that could be used as a checklist for monitoring systems. Besides the improvement of basic properties like scalability, they suggest to focus more on standardization and energy efficiency.

V. Conclusion

In this paper, we have described a scalable monitoring system for cloud computing that is currently developed in the *EASI-CLOUDS* project. The monitoring data are locally collected from various suppliers, but integrated in a global tree structure that provides location-transparent access to them via a generic interface. To potentially adapt the monitoring data to the requirements of the subscribers, the system has a context-processing platform for data storage and preprocessing.

The *EASI-CLOUDS* project develops a cloud concept in which IaaS, PaaS and SaaS can be offered as billable services with SLAs. In order to manage the hardware resources and to avoid SLA violations, the scalable monitoring system is an essential component. Our future work will focus on the integration of further data suppliers and on an easily extendible base implementation of the context-processing platform.

Acknowledgement

This work has been funded in part by the Federal Ministry of Education and Research (BMBF) under contract no. 01IS11021 (project EASI-CLOUDS).

References

[1] G. Aceto, A. Botta, W. de Donato, and A. Pescape. Cloud monitoring: Definitions, issues and future directions. In *Cloud Networking (CLOUDNET), 2012 IEEE 1st International Conference on*, pages 63–67, 2012.

[2] M. Andreolini, M. Colajanni, and S. Tosi. A software architecture for the analysis of large sets of data streams in cloud infrastructures. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 389–394, 2011.

[3] C. Canali and R. Lancellotti. Automated clustering of vms for scalable cloud monitoring and management. In *Software, Telecommunications and Computer Networks (SoftCOM), 2012 20th International Conference on*, pages 1–5, 2012.

[4] CloudFoundry. RabbitMQ + Cloud Foundry: Cloud Messaging that Just Works. http://blog.cloudfoundry.com/2011/08/09/rabbitmq-cloud-foundry-cloud-messaging-that-just-works/, 2011.

[5] S. A. de Chaves, R. B. Uriarte, and C. B. Westphall. Toward an architecture for monitoring private clouds. *IEEE Communications Magazine*, 49(12):130–137, 2011.

[6] O. Dohndorf, J. Krüger, H. Krumm, C. Fiehe, A. Litvina, I. Lück, and F.-J. Stewing. Lightweight policy-based management of quality-assured, device-based service systems. In *Proceedings of the IEEE 24th International Conference on Advanced Information Networking and Applications (AINA 2010)*, pages 526–531, Perth, Australia, 2010. IEEE Computer Society.

[7] O. Dohndorf, J. Krüger, H. Krumm, C. Fiehe, A. Litvina, I. Lück, and F.-J. Stewing. Tool-supported refinement of high-level requirements and constraints into low-level policies. In *Proceedings of the 2011 IEEE International Symposium on Policies for Distributed Systems and Networks (Policy 2011)*, Pisa, Italy, 2011. IEEE Computer Society.

[8] C. Fiehe, A. Litvina, I. Lück, F.-J. Stewing, O. Dohndorf, J. Krüger, and H. Krumm. Policy-gesteuertes management adaptiver und gï¿$\frac{1}{2}$tegesicherter dienstesysteme im projekt osami. In *Proceedings 154 - Informatik 2009 Im Focus das Leben*, pages 970–983, Lübeck, Germany, 2009. Gesellschaft für Informatik / Verlag Köellen.

[9] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, 2000. AAI9980887.

[10] S. Karapetkov. Polycom uc intelligent core: Scalable infrastructure for distributed video. Technical report, 4750 Willow Road, Pleasanton, CA 94588, 2010.

[11] B. Koï¿$\frac{1}{2}$nig, J. Alcaraz Calero, and J. Kirschnick. Elastic monitoring framework for cloud infrastructures. *Communications, IET*, 6(10):1306–1315, 2012.

[12] M. F. S. M. Suhail Rehman, Mohammad Hammoud. Votus: A flexible and scalable monitoring framework for virtualized clusters. In *In Proceedings of The 3rd International Conference on Cloud Computing and Science (CloudCom 2011), Athens, Greece, December 2011*, pages 1–4, 2011.

[13] Nagios. Nagios xi-maximizing xi performance. Technical report, Saint Paul, MN, USA, 2012.

[14] Nagios. Nagios-distributed monitoring solutions. Technical report, Saint Paul, MN, USA, 2013.

[15] Open Mobile Alliance. OMA Device Management Tree, March 2012.

[16] Oracle. Broker Clusters: Scalability and Availability. http://docs.oracle.com/cd/E19879-01/821-0028/ggsbb/index.html, 2010.

[17] OSGi Alliance. OSGi Service Platform Service Compendium - Dmt Admin Service Specification, January 2012.

[18] B. Project. bischeck dynamic and adaptive thresholds for Nagios. http://www.bischeck.org/?page_id=435.

[19] K. Ren, J. López, and G. Gibson. Otus: resource attribution in data-intensive clusters. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 1–8, New York, NY, USA, 2011. ACM.

[20] M. Wang, V. Holub, T. Parsons, J. Murphy, and P. O"Sullivan. Scalable run-time correlation engine for monitoring in a cloud computing environment. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 29–38, 2010.

[21] C. Wickramarachchi, S. Perera, S. Jayasinghe, and S. Weerawarana. Andes: A highly scalable persistent messaging system. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 504–511, 2012.