Towards Holistic Multi-Tenant Monitoring for Virtual Data Centers

Peer Hasselmeyer, Nico d'Heureuse
NEC Laboratories Europe, NEC Europe, Ltd.
69115 Heidelberg, Germany
{Peer.Hasselmeyer,Nico.dHeureuse}@neclab.eu

Abstract—**Cloud computing becomes increasingly prevalent for outsourcing IT functions. The basic feature of offering virtual data center slices to customers has been in use for some time now. So far, customers only get the raw resources, with only little insight and control of their resources. But to let customers build reliable services on top of the rented infrastructure, they need adequate monitoring and control capabilities. In the future, we expect operators to offer such functions to their customers.**

In this paper, we introduce our approach towards offering a holistic monitoring system to data center customers. It offers generic monitoring information propagation and storage covering various types of resources (network, servers, and applications), all kinds of monitoring information, and all tenants. As virtualized data centers are usually large and multi-tenant, our solution is built with these properties in mind.

Keywords-cloud computing, monitoring, multi-tenancy, scalability

I. INTRODUCTION

Cloud computing [1] in its form of Infrastructure-as-a-Service (IaaS) [2] is increasingly becoming a viable solution for outsourcing various IT-related functions. The easy and flexible addition and removal of resources make it an attractive solution for deploying and offering services with unknown and fluctuating traffic volume. The complete removal of the burdensome administrative tasks for both hardware and software maintenance and the associated savings in operating expenses make cloud computing attractive to companies who want to focus on their core business rather than on resource management.

Cloud providers offer IaaS with the help of large data centers (DCs). On top of the physical hardware, a virtualization layer is introduced which slices the physical servers into smaller instances which are then offered to customers. Multiple virtual servers can be hosted on one physical machine, allowing providers to sell physical machines multiple times thereby increasing revenue as well as utilization of the individual machines.

Cloud providers have implemented automated procedures for provisioning new virtual servers and retiring unused ones allowing them to quickly adapt to customers' changing needs. The processes are based on customer self-service, meaning that customers are heavily involved in the process of acquiring, releasing, and configuring resources.

A large part of the customers of cloud providers are service providers who use cloud resources to provide value-added services to their customers. With cloud computing, service providers can focus on their core business of creating services and do not have to deal with the peculiarities of efficiently operating hardware.

Service providers usually offer services to their customers with certain Quality-of-Service (QoS) guarantees specified in Service Level Agreements (SLAs). To make this possible, providers need to know the capabilities of the infrastructure they base their services on. With in-house operation of data centers, this is easy as the hardware, the set-up, the configuration, and the operating procedures are known. In the case of outsourcing to a cloud provider, such knowledge is not available to service providers, as cloud providers treat this as corporate secrets. Another solution for assessing and guaranteeing QoS is therefore needed.

A possible solution is the specification of SLAs for the IaaS resources rented from cloud providers. Service providers can translate the requirements of their services and applications to requirements on the IaaS infrastructure. Such requirements are then specified in contracts with the cloud provider.

Cloud providers cater for these requirements by offering their services in a variety of quality levels and by providing SLAs for them. At present, the possibilities for service providers to monitor adherence to SLAs is severely restricted as monitoring of parameters relevant to SLAs is usually not offered to clients of cloud services.

With the growing sophistication of applications running on cloud infrastructures, monitoring and control of virtual data centers will become more important and ultimately an indispensible feature. As monitoring is a prerequisite for meaningful control, monitoring functionality is needed for both assessing the current status of the resources and as a basis for control.

For complete insight into the status of their services, service providers need information about all layers of their system, including the application, the virtual servers, and the virtual network. For simplicity reasons, all this information is ideally delivered via a single unified monitoring fabric. Also, cloud providers do not want to support separate monitoring infrastructures for every tenant. A single scalable fabric, sliced into virtual chunks allows reaping benefits from economies of scale.

This paper introduces our approach toward providing a monitoring infrastructure for cloud systems. It addresses the particular requirements of cloud systems and offers multi-tenancy, scalability, dynamicity, simplicity and comprehensiveness. The system is based on data stream management systems to efficiently transport, filter, and aggregate monitoring information.

The remainder of the paper is structured as follows. Section II identifies the particular requirements that are relevant to cloud monitoring. Section III describes the architecture of the monitoring infrastructure that we developed, while section IV assesses its suitability to fulfill the requirements outlined before. Section V provides details on our implementation of the architecture. Section VI discusses some related work while section VII concludes the paper.

II. Requirements for a Cloud Monitoring System

Providing monitoring information of the plurality of virtual data centers to the respective tenants puts some requirements on the monitoring infrastructure that are particular to this environment. In the following, the main requirements that make this domain different from "regular" data center monitoring are detailed.

Multi-tenancy. The monitoring infrastructure must naturally be able to deal with monitoring information belonging to the different customers (or, "tenants") of the data center. On one hand, tenants must not receive monitoring information that is addressed to other customers ("isolation"). The reason for this is the privacy requirement of the monitored information which is also regulated by data-privacy laws.

On the other hand, some information is to be propagated to multiple tenants. As an example, imagine a problem with a physical server which affects all the virtual machines running on it. Such information needs to be propagated to all the tenants whose virtual machines are affected.

Scalability. A holistic (in terms of monitoring information and in terms of tenants) monitoring solution must be scalable in various aspects. It must scale to large numbers of monitoring agents, of event notifications, of tenants, of resources (virtual and physical ones; servers, network elements, and applications), and of types of monitoring information. Ideally, the solution scales out infinitely, by adding pieces of equipment (servers and/or network). Intuitively, the number of event notifications increases with the number of agents, tenants, and resources. But as agents provide their measurements at different intervals, this is only a loose correlation.

Dynamism. Multi-tenant monitoring systems must support the dynamism that is inherent in multi-tenant data centers. The dynamism stems from the quick and frequent addition and removal of tenants to/from the data center. Moreover, the assignment of resources to tenants is constantly changing as well. Beyond that, tenants can change the set of monitored resources at will.

The result of these three drivers is constant change in the configuration of the monitoring system. A monitoring solution must be at least as agile as the monitored system.

Simplicity. The monitoring system should be simple in two aspects: first, the interface to the monitoring system must be easy to understand, use, and code against. Second, the system must be easy to install and maintain – for both, the data center operator and its customers.

Simplicity in the application programming interface (API) is desired as it eases the job of developers that create monitoring and control solutions. Making it easy for developers to implement solutions is an important virtue to increase take-up of the interface and therefore the monitoring solution.

Administrative simplicity is important to the data center operator. The easier it is to install and maintain a software solution, the less reluctant he will be to install such solution in his data center. In addition, less maintenance effort means less human resources to be committed, making the operation of the monitoring solution cheaper and giving the opportunity to either increase income or to reduce cost charged to customers.

Comprehensiveness. Closely related to the requirement of simplicity is the requirement for comprehensiveness. It relates to the need that one single monitoring infrastructure should be usable for all kinds of monitoring information, no matter what resources it relates to and what meaning it conveys. Comprehensiveness applies to multiple aspects: data types, notification source, and tenants. The benefits of having a holistic monitoring system are twofold. First, developers only need to learn a single monitoring API, independent of what kind of monitoring information is accessed in their applications. Second, the data center operator needs to deploy and maintain only one single monitoring infrastructure, not multiple separate ones for each monitoring domain (network, IT, applications).

III. Cloud Monitoring System Architecture

The requirements listed in the previous chapter can be fulfilled in different ways. We developed a solution that is based on one single monitoring infrastructure (at least conceptually). In this section we introduce the functional building blocks as well as the data format of the event notifications. These two are the enablers of our monitoring infrastructure and are key to fulfilling the requirements stated before.

A. Functional Architecture

The functional architecture is shown in Figure 1. A main building block of the architecture is a generic event notification propagation system, called a "message bus" in the figure. The component can be realized in different ways. We chose to use a data stream management system here, as described later on in section V.

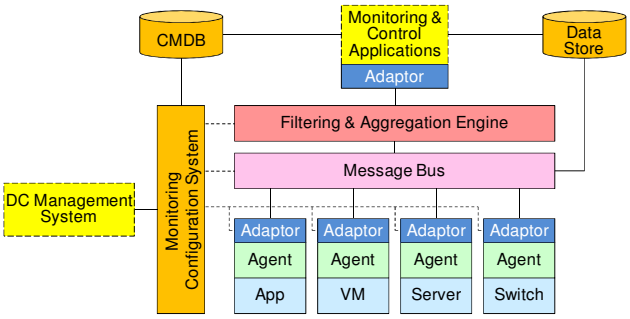

Figure 1. Monitoring System Functional Architecture

Closely related to the message bus is a filtering and aggregation engine. As the name suggests, it filters event notifications based on certain criteria and in particular aggregates notifications. The aggregation facility is mostly used to aggregate streams of individual notifications over time, meaning that it creates coarse-grained streams of notifications from finer grained ones. For example, it can be used to create notifications of CPU utilization with a granularity of one minute from a stream of CPU utilization measurements with the granularity of one second. Similarly, it is used to aggregate streams of measurements of the same type originating from different agents into one aggregated stream. For example, CPU utilization measurements from different servers can be accumulated into one stream expressing the aggregated CPU utilization of the server farm.

Every monitorable resource in the data center has one or more agents attached to it. The agents are the source of all monitoring information. They measure certain parameters and periodically or on-demand pass this information on to the message bus. Once on the message bus, the notifications are filtered and aggregated and, depending on the configuration, they are propagated to specific monitoring and control applications and are stored for later use in a data store.

The monitoring and control applications are external to the monitoring infrastructure and are supplied and operated by either the data center operator or the tenants. These applications can therefore implement behavior that is specific to particular tenants of the data center and contain business and operational logic that distinguishes a tenant from its competitors. As these applications are not part of the monitoring infrastructure they are not described in any detail here. Only the interface between the applications and the infrastructure is part of this paper.

Both agents and monitoring applications are decoupled from the message bus by an adaptation layer (shown as "adaptor" in the figure). This adaptation layer fulfils multiple important functions. First, it abstracts from the actual core monitoring system implementation and thereby insulates the agents and applications from the particular implementation technology used for the message bus and aggregation engine (which can be implemented using different technologies). Second, it adds functionality that makes the system usable in multi-tenant data center environments. In particular, it adds functionality for dealing with multiple tenants. This includes the registration of the agents with the monitoring configuration system. Agents announce their capabilities and the resources they monitor to the configuration system. The information about the agents is stored in a Configuration Management Database (CMDB). Applications can query the CMDB, e.g., to find out which agents are available for a certain tenant. Furthermore, the CMDB stores all other configuration-related parameters of the system. The CMDB therefore acts like a service repository for registering and querying monitoring agents and their capabilities.

Following the *Simplicity* principle, the configuration of an agent should be – from a user's point of view – as simple as possible. Generally, the agents have no information about their own location or about the tenant(s) they are associated with. The monitoring system, on the other hand, needs to associate each data message with the sending agent as well as with the corresponding tenants. Thus, there has to be a way for the agents to acquire information about their location and to identify themselves in a unique way. This identification functionality is provided by the adaption layer: when an agent is started, the adaptation layer requests a unique identifier from the monitoring system. Depending on the security requirements of the system, authentication and authorization mechanisms may be invoked at this point in time. The system will then generate such an identifier and associate it with the corresponding tenants. This association can be done either using the previously invoked authentication step or by determining the physical/virtual location of the agent (e.g., by inspecting the agents IP address, or by interacting with the hypervisor).

Longer-term storage of events (e.g. longer than one hour) happens in a data store, which can be a regular data base or a cloud-like storage infrastructure, such as BigTable [4]. Data stored here is used for later analysis and for trend visualization using an appropriate GUI. Similar to the monitoring and control applications, the GUI can be provided by different parties and might be specific to particular tenants.

The monitoring system is subject to configuration. The configuration system takes care of configuring the filtering and aggregation system and the agents on monitored resources. It performs multiple actions with regard to the agents. First, it maintains the list of available agents in the CMDB. Second, it configures the agents (as found in the CMDB) to provide monitoring information as needed. In particular, the frequency of periodic measurement updates and thresholds for event-based notifications are configured. In addition, the configuration system tells the agents about the tenants the monitoring information should be sent to. Third, the configuration system controls the filtering and aggregation system. In particular, it configures the aggregation intervals for accumulating measurement streams.

To perform its duties, the monitoring configuration system needs information about tenants, their resource assignments, and their preferences. As these pieces of information are primarily needed for other management duties, they are already kept outside the monitoring system by the operator's data center management system. The monitoring configuration system therefore needs an interface to access such information.

B. Notification Data Format

The data format of event notifications is shown in Figure 2. Besides the actual monitoring information to be conveyed, there are five fields containing meta-data.

sourceType	String
sourceID	String
dataType	String
tenants	String
timeStamp	dateTime
data	XML

Figure 2. Monitoring Information Data Structure

The *sourceType* field contains a unique identifier of what kind of information is transferred in this message. It identifies the syntax of the data conveyed as well as their semantics. The latter is important for applications to filter out the information that they understand and that they are able to process. Although the *sourceType* field is a string parameter, we use a URI (uniform resource identifier) syntax here. Examples are: `http://example.org/NetworkLoad`, `http://example.org/mail/MessagesPerSecond`.

The *sourceID* field uniquely identifies the origin of the monitoring information. The information contained in here must be unique within the scope of the associated *sourceType*. For physical servers, for example, the IP address could be a reasonable source identifier within a data center. For an application deployment it might consist of the application name and an instance identifier, if multiple copies of the application are running.

The *dataType* field is related to the *sourceType* field. It identifies the syntax of the monitoring data transferred inside this notification. As described above, *sourceType*s already denote the syntax of the conveyed data. The problem is that recipients need to know a particular *sourceType* in order to infer the syntax. As measurement probes can have arbitrary *sourceType*s, we expect applications to only know and understand a very limited set of these types. Creating generic monitoring applications is hard to impossible without the knowledge of the actual syntax of the data. We therefore introduced the *dataType* field which allows the specification of the syntax of the transferred data. As oftentimes simple data types (like integers, strings, and floats) are used for monitoring measurements, generic monitoring applications (e.g. visualizations) can be created that can handle any kind of monitoring information as long as it can be expressed with such a simple (or complex, but well-known) type. Examples are: `xsd:float`, `xsd:int`.

The *tenants* field contains a semicolon-separated list of tenants that this monitoring information relates to. As monitoring information is only allowed to flow to authorized recipients, this field mainly restricts visibility of monitoring information to the tenants listed in this field.

The *timeStamp* field denotes the time of creation of a notification message. These time stamps can be used for ordering events, at least within one particular stream. Ordering of messages across streams is possible up to a certain time resolution, as we assume resource clocks to be synchronized.

Two examples of complete monitoring information packets are shown in Figure 3.

sourceType	http://example.org/NetworkLoad
sourceID	192.168.1.20
dataType	xsd:float
tenants	customer86
timeStamp	2009-12-10T05:52:19.358
data	<value>0.21</value>

sourceType	http://example.org/mail/MessagesPerSecond
sourceID	mailserver-03
dataType	xsd:int
tenants	operator
timeStamp	2010-01-04T14:28:05.734
data	<value>83</value>

Figure 3. Examples of Monitoring Packets

IV. Cloud Monitoring Architecture Assessment

The architecture addresses the requirements in the following way:

Multi-tenancy. Every tenant's monitoring information flows through the same stream management system, but the information is only to be seen by its intended recipient(s). The system therefore has to make sure that information about the intended recipient(s) is transferred with the messages and that messages are appropriately filtered before being passed to user applications. Isolation in terms of tenant visibility is thereby achieved.

Such filtering is performed by the message propagation system before delivering messages to their recipients. The data format as described before contains a field that denotes the intended recipients. This field is stripped from the notification before delivery to the recipients in order to avoid recipients being able to learn about other recipients of a particular message.

The handling of tenant information is therefore completely shielded from the tenants, making the multi-tenancy property invisible to the tenants.

Scalability. As described before, scalability is required in multiple domains: the number of monitoring agents, number of event notifications, number of tenants, number of resources, and number of types of monitoring information.

An arbitrary number of monitoring agents is supported. As the location of the agents is of no importance to the monitoring system, the agents can be put in locations that are convenient for ensuring scalability. As they can be placed together with the monitored resources, the number of agents can scale with the number of resources.

The monitoring system potentially scales with the number of event notifications. Although the data stream management system could be a limiting factor to scalability, it is possible to increase the resources (in terms of capacity of a single machine and in terms of boxes) dedicated to it and to distribute the monitoring load across the involved resources. Spreading the

load can happen in a number of fashions. One possibility is to assign different tenants to different instances of the monitoring infrastructure. Each instance is then independent of the other instances. Another possibility is to create multiple monitoring infrastructure instances that are distributed according to locality inside the data center (e.g., one per rack or per aisle). Other approaches are possible and which one is best needs to be further investigated.

Dynamism. Addition and removal of tenants and resources is directly supported by the infrastructure. As described before, tenants are identified by a character string. The monitoring system does not need to know the set of identifiers currently in use (i.e., the set of tenants using the data center) or what a particular identifier means. Addition and removal of tenants is therefore handled without further work on the side of the notification propagation system. What does need particular attention is the configuration of the storage and aggregation system. As tenants want to store and aggregate monitoring information, the system needs to be appropriately configured which in turn requires information about the set of known tenants. In particular, the withdrawal of a tenant needs to be propagated in order to remove the configuration of the aggregation system for that particular tenant.

Assignment and withdrawal of resources to/from a tenant is addressed in the architecture by a number of mechanisms. To the monitoring system, assignment/withdrawal means the addition or removal of monitoring agents. Propagating information from these new/removed agents is supported as the monitoring system simply propagates notifications from any agent that wishes to do so. In addition to notification propagation, appropriate agents need to be instantiated on newly added resources. The monitoring configuration component therefore needs to be informed of added resources. It then takes the appropriate actions to install and/or activate the agents according to the wishes of the tenant the resource was assigned to. Similar activities might be needed upon withdrawal of a resource from a tenant.

The set of monitoring data types flowing inside the monitoring system can be dynamic as well, as each agent can have its own data type associated with its monitoring information. The monitoring system does not need to know the current set of data types used as it propagates any such information and does not do any further inspection of the monitoring data. Aggregation of monitoring information can only be performed for well known data types. Messages conveying data of other, unsupported types are ignored.

Simplicity. The architecture provides a simple, combined interface for all kinds of monitoring information. Publishing as well as receiving event notifications is handled via small interfaces that abstract from the peculiarities of the implementation technology. Developers of monitoring agents as well as monitoring applications do therefore not need to be concerned with the particular monitoring system implementation.

Administrative simplicity is potentially given, as there is only one system to install and maintain. As tenants are given control over the configuration of their slice of the monitoring system, the data center operator is saved from dealing with such configuration. The data center operator is only concerned with the installation of the basic monitoring infrastructure (and potentially with the configuration of his slice of the monitoring system). This work can have different degrees of complexity depending on the size of the installation and how difficult a monitoring system split (with respect to scalability) has been chosen.

Configuration simplicity for the individual tenants is another important aspect. Depending on the size of the virtual data center a tenant is operating, his notion of simplicity varies. While for a small set-up, an intuitive graphical user interface is probably the simplest solution; large installations need more automation and a programming interface is most likely the best solution. The architecture therefore provides such a programming interface on which graphical user interfaces can be built according to the requirements of the data center operator and its tenants.

Comprehensiveness. We require comprehensiveness in three domains: resources, data, and tenants. The architecture caters for comprehensiveness by its use of XML inside event notifications which allows arbitrary monitoring data to be encoded and transferred. The tagging with a type identifier allows applications to filter messages and to determine whether they can make sense of the received information.

As described above in the paragraphs on scalability and dynamism, the architecture allows for arbitrary notification sources, therefore covering the whole universe of monitoring agents and their associated resources. As agents can belong to arbitrary tenants, the complete set of tenants is inherently supported by the architecture as well.

Alternative Solutions. A straight-forward alternative architecture for a monitoring system is the use of a separate monitoring installation for every tenant. If set up correctly, such a solution directly solves the problems of multi-tenancy and scalability (at least in the number of tenants, not necessarily in the number of events/resources). We did not opt for such a solution as it requires the data center operator to manage separate monitoring infrastructures for all tenants. We believe that such an approach will incur too high an overhead than to be economically sustainable.

V. CLOUD MONITORING SYSTEM IMPLEMENTATION

The proposed system is currently being implemented. For the message bus and filtering and aggregation engine components we chose to use a data stream management system (DSMS) [3]. Data stream management systems are software components that are optimized for fast processing of streams of data. In particular, they provide mechanisms for continuously querying such streams. The most commonly used query functions include selection of events that fulfill particular requirements and aggregation of multiple events.

What makes data stream management systems so attractive for monitoring are these query capabilities. Applied to monitoring, data stream management systems allow for aggregating discrete monitoring events over time and space and for extracting particular notifications from the stream. The two architectural components message bus and filtering and

aggregation engine are therefore realized by a single software component – the DSMS.

For the longer-term data store, we are currently using a regular data base management system.

Our monitoring system is implemented in Java. Accordingly, all the adaptation classes are using the Java language and components interacting with the monitoring system, namely agents and monitoring and control applications, need to be implemented, at least in part, in the Java language.

We have implemented an agent framework that allows for the periodic retrieval of measurements from arbitrary probes and their propagation to the message bus. The framework also has a management interface that allows the configuration of the frequency with which particular measurements are taken.

The interfaces used to interact with the monitoring system are shown in Figure 4. The anchor to the monitoring system is the Monitoring class which grants access to objects for publishing and which allows registration for receiving notifications.

```
public class Monitoring {
    public static Publisher getPublisher() {}
    public static CallbackRegistration register(
                    String sourceType, String sourceID,
                    String dataType,
                    NotificationCallback callback) {}
}

public interface Publisher {
    public void publish(String sourceType, String sourceID,
                        String dataType, String data);
}

public interface NotificationCallback {
    public void notify(String sourceType, String sourceID,
                       String dataType, String data);
}
```

Figure 4. Notification Publication and Reception Interfaces

All interfaces are similar in the sense that they accept similar arguments. Unlike the other functions, the registration method allows the use of patterns to match a broader scope of notifications. Event notification is realized with a callback pattern in which notifications are signaled using the notify() method. The CallbackRegistration interface which is not shown in the above figure only contains a cancel() method to remove notification registrations.

Please note the absence of tenant information from the method signatures. One important aspect of the monitoring system is that the information on tenants inside monitoring packets must be correct and reliable in order to prevent information privacy breaches. The two properties are ensured by not granting tenants and agents any access to that information. Tenant handling is completely concealed inside the monitoring system. Observation of tenant information restrictions is automatically enforced by the filtering and aggregation engine by creating queries that do not allow the release of tenant information.

On the agent side, monitoring information packets are tagged with the intended recipients. This is easy for measurement agents that run under the sole control of the data center operator. Code running under the control of the data center customer can always be tampered with and can therefore supply false tenant information. Authentication can be used to counter this.

Tenant information can be supplied in different ways to the agents. The two ways that we are using in the current implementation are as follows. First, agents running under client control can read the tenant information from the run time environment. Such information is read automatically by the adaptation code and added to all outgoing monitoring information packets. Agents that are running external to the monitored system (e.g., an SNMP agent) are configured with tenant information via their management interfaces.

VI. Related Work

Data stream management systems [3] have been proposed for network monitoring and fault detection before. Some promising results are presented in [5], [6], and [7]. The main focus of these systems is network traffic analysis. None of them deals with a multi-tenant environment. Our work extends monitoring systems based on DSMS with the ability to handle multiple tenants and arbitrary data.

Nagios [8] is a popular systems management environment that can be extended to visualize monitoring information from various sources, including network, servers, and applications. It lacks support of dynamic, on-the-fly reconfigurations required for agile environments. Also, scalability issues and lack of sophisticated multi-tenant support prevent *Nagios* from being an appropriate choice for a cloud data center.

Some public cloud providers (e.g., Rackspace [9], Amazon [10]) already offer basic monitoring services. Usually, only those parameters are monitored which are required for billing purposes such as network traffic or CPU hours. Amazon's *CloudWatch* service [11] additionally provides monitoring of the deployed virtual infrastructure with a somewhat finer granularity (e.g. request count for load balancers). However, even *CloudWatch* does not allow customers to extend the monitoring platform, i.e., to stream their own events into the monitoring framework.

VII. Conclusion

In this paper we describe our approach towards a cloud monitoring system that enables customers to get insight into their rented virtual resources. We propose a monitoring infrastructure that was designed with scalability, multi-tenancy, dynamism and simplicity as major design goals. We describe the architecture of the monitoring infrastructure in terms of functional building blocks and showed how it addresses the design goals.

We are currently implementing the architecture with a data stream management system as the main event propagation, filtering, and aggregation component. We developed adaptation code that makes it easy to interact with the monitoring system. Our choice of loose coupling and the design of the associated data structures give the system a high degree of flexibility and

dynamism in terms of tenants, resource assignment, and monitoring agents.

We believe that the architecture provides all the desired features that make it usable and useful in a cloud environment. As the implementation of the system is still on-going, we do not have concrete results yet. Once the infrastructure is in place, we will perform scalability and usability tests in order to assess the degree of fulfillment of the requirements outlined in this paper.

REFERENCES

[1] M. Armbrust, et al., "Above the Clouds: A Berkeley view of Cloud Computing," Technical Report No. UCB/EECS-2009-28, University of California at Berkeley, USA, February 10, 2009.

[2] P. Mell, T. Grance, "The NIST Definition of Cloud Computing," version 15, National Institute of Standards and Technology, Information Technology Laboratory, October 7, 2009. Available online at: http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc

[3] J. Hyde, "Data in Flight," ACM Communications, vol. 53, no. 1, pp. 48-57, January 2010.

[4] F. Chang, et al., "Bigtable: A Distributed Storage System for Structured Data," in Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation, November 6-8, 2006, Seattle, WA, USA.

[5] S. Babu, L. Subramanian, J. Widom, "A data stream management system for network traffic management," in Proceedings of the 2001 Workshop on Network-Related Data Management (NRDM 2001), May 25, 2001, Santa Barbara, CA, USA.

[6] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk, "Gigascope: a stream database for network applications," in Proceedings of the 2003 ACM SIGMOD international conference on management of data, June 9-12, 2003, San Diego, CA, USA.

[7] J. Guo-quan, D. Bo, Z. Xiao-yi, D. Kun, "Producing synoptic data structure for high speed data streams in telecommunication network management," in Proceedings of 4th International Conference on Computer Science & Education (ICCSE '09), July 25-28, 2009, Nanning, China.

[8] Nagios Enterprises, LLC., Nagios Homepage, http://www.nagios.org/.

[9] Rackspace, US Inc., The rackspace cloud, http://www.rackspacecloud.com

[10] Amazon, Amazon Web Services (AWS), http://aws.amazon.com.

[11] Amazon, Amazon CloudWatch, http://aws.amazon.com/cloudwatch