A Publish/Subscribe Middleware for Dependable and Real-time Resource Monitoring in the Cloud *

Kyoungho An, Subhav Pradhan, Faruk Caglar, Aniruddha Gokhale
Institute for Software Integrated Systems (ISIS)
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
{kyoungho.an, subhav.m.pradhan, faruk.caglar, a.gokhale}@vanderbilt.edu

ABSTRACT

Providing scalable and QoS-enabled (*i.e.*, real-time and reliable) monitoring of resources (both virtual and physical) in the cloud is essential to supporting application QoS properties in the cloud as well as identifying security threats. Existing approaches to resource monitoring in the cloud are based on web interfaces, such as RESTful APIs and SOAP, which cannot provide real-time information efficiently and scalably because of a lack of support for fine-grained and differentiated monitoring capabilities. Moreover, their implementation overhead results in a distinct loss in performance, incurs latency jitter, and degrades reliable delivery of time-sensitive information. To address these challenges this paper presents a novel lighter weight and scalable resource monitoring and dissemination solution based on the publish/subscribe (pub/sub) paradigm. Our solution called SQRT-C leverages the OMG Data Distribution Service (DDS) real-time pub/sub middleware, and uses effective software engineering principles to make it usable with multiple cloud platforms. Preliminary empirical results comparing SQRT-C with contemporary web-based resource usage monitoring services reveals that SQRT-C is significantly better than the conventional approaches in terms of latency, jitter and scalability.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*real-time, fault-tolerance, availability*

General Terms

Monitoring, Reliability, Performance

*This work was supported in part by NSF awards CAREER/CNS 0845789 and SHF/CNS 0915976. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SDMCMM'12, December 3–4, 2012, Montreal, Quebec, Canada.
Copyright 2012 ACM 978-1-4503-1615-6/12/12 ...$15.00.

Keywords

Resource Monitoring, Cloud Computing, Pub/Sub Middleware

1. INTRODUCTION

Cloud computing is a distributed computing paradigm that provides massively scalable resources as services to customers who reside outside the cloud [1]. As businesses move towards leveraging resources from the cloud rather than procuring and maintaining them in-house, cloud customers expect the same quality of service (QoS) for their services from the cloud as they did when the applications were hosted in-house at the customer's premises. Even the US Department of Defense is moving towards hosting their mission-critical applications in the cloud [2], which will require the cloud to assure stringent QoS properties.

In other words, the cloud must be dependable in that it must support application QoS despite fluctuations in availabilities of cloud resources and their failures, adapt to changes in application workloads, and assure secure operations. Although solutions for autoscaling, fault tolerance and security mechanisms are needed to address these challenges, at the core of all these solutions lies the need for accurate, timely, dependable, and scalable monitoring of both the physical and virtualized resources of the cloud. This becomes all the more important as the Resource-as-a-Cloud (RaaS) [3] model becomes more prevalent.

Related Work: Contemporary compute clusters and grids have provided special capabilities to monitor the distributed systems via frameworks, such as Ganglia [4] and Nagios [5]. Additionally, [6] provides a comparative study of Pub/Sub middleware for real-time grid monitoring in terms of real-time performance and scalability. According to [7], one of the distinctions between grids and clouds is that cloud resources are more abstracted and virtualized compared to grid resources. However, these frameworks are structured primarily to monitor physical resources only, and not a mix of virtualized and physical resources. Even though some of these tools have been enhanced to work in the cloud, *e.g.*, virtual machine monitoring in Nagios[1] and customized scripts used in Ganglia, they still do not focus on the timeliness and reliability of the dissemination of monitored data that is essential to support application QoS in the cloud.

In other recent works, [8] presents a virtual resource monitoring model while [9] discusses cloud monitoring architecture for private clouds. Although these prior works describe

[1]`http://people.redhat.com/~rjones/nagios-virt`

cloud monitoring systems and architectures, they do not provide experimental performance results of the models, such as overhead and response time. Consequently, we are unable to determine their relevance to host mission-critical applications in the Cloud. Latency results using RESTful services are described in [10], however, they are not able to support diverse and differentiated service levels for cloud clients.

Paper Contributions: We surmise that the publish/subscribe (pub/sub) [11] paradigm can overcome the limitations with existing monitoring and dissemination mechanisms that use RESTful APIs. Moreover, since dependability (*i.e.*, performance and timeliness) in information dissemination is a key need, a specific form of pub/sub supported by the OMG Data Distribution Service (DDS) [12] for data-centric pub/sub, is a promising technology to disseminate resource monitoring data of virtual machines on physical nodes both scalably and in real-time.

This paper describes our DDS-based solution called *Scalable and QoS-enabled virtual resource monitoring system for Real-Time applications in Clouds* (SQRT-C). SQRT-C is compatible with cloud software for IaaS, such as OpenNebula, Eucalyptus, and OpenStack–a property it acquires by exploiting the *libvirt* library that interacts with a hypervisor to retrieve information about virtual machines.

The remainder of this paper is organized as follows. Section 2 describes the system architecture and implementation of SQRT-C, Section 3 discusses experimental results evaluating SQRT-C, and finally Section 4 offers concluding remarks alluding to future work.

2. DESIGN OF SQRT-C

This section presents SQRT-C, which builds on top of libvirt for accessing resources of virtual machines, and OMG DDS for real-time dissemination of the virtual resource information. To better understand our solution, we present an overview of DDS and describe the design considerations in adopting DDS. Subsequently, we present the architecture of SQRT-C.

2.1 Overview of OMG DDS

The OMG Data Distribution Service (DDS) specifies a layered architecture comprising three layers. Two of these layers are useful in making DDS a promising design choice for use in the scalable and timely dissemination of resource usage information in a cloud platform. One of the layers is called Data Centric Publish/Subscribe (DCPS), which provides a standard API for data centric, topic-based, real-time pub/sub [13]. It provides efficient, scalable, predictable, and resource-aware data distribution capabilities. The DCPS layer operates over another layer that provides a DDS interoperability wire protocol [14] called Real-Time Publish/Subscribe (RTPS).

One of the key features of DDS when compared to other pub/sub middleware is its rich support for QoS offered at the DCPS layer. DDS provides the ability to control the use of resources such as network bandwidth and memory, and non-functional properties of the topics, such as persistence, reliability, timeliness, and others [15].

2.2 Design Considerations in Adopting DDS for Cloud Platforms

Despite the promise shown by DDS, it is not straightforward to deploy DDS within a cloud platform and expect the real-time dissemination of resource usage information to interested entities. Thus, a number of design considerations in architecting SQRT-C must be accounted for in adopting DDS as described below:

1. **Accessing physical resource usage in a virtualized environment:** Cloud customers cannot access physical machines of a cloud to obtain their resource usage data; rather only virtual resources in data centers are granted as a service to the customers. Thus, customers should be able to retrieve accurate resource usage at the level of physical resources. Moreover, any solution to address this must be non-invasive to the deployed cloud platform so that it can work with a range of cloud software.

2. **Managing DDS entities:** A large data center in a cloud will comprise a large number of physical resources and an even larger number of virtualized resources. Similarly, a large number of applications may be hosted simultaneously in a data center at any given time. In turn this entails the presence of a large number of DDS publishers (data writers) and subscribers (data readers), and hence a large number of DDS topics and topic instances that must be managed and disseminated.

3. **Shielding cloud users from complex DDS QoS configurations:** The desired timeliness and dependability QoS properties for resource usage dissemination can be controlled by configuring the publishers and subscribers with the DDS QoS policy settings. However, it is not trivial for the subscribers to thoroughly understand the diverse range of DDS QoS policies and avoid conflicts between them. Ideally, the subscribers (who are the cloud customers) should not have to deal with these issues and the presence of DDS should be completely masked from them.

2.3 Architecting SQRT-C

Figure 1 illustrates the SQRT-C architecture. We have borrowed terminology, such as Cluster Node and Front-end Node, from the OpenNebula [16] open source cloud platform to represent the physical computing entities inside the cloud. SQRT-C uses the DDS-based pub/sub technology to disseminate resource usage information for virtual resources from the source (*i.e.*, publishers) to the sinks (*i.e.*, the subscribers) while also supporting the QoS requirements on the dissemination of the monitored information.

2.3.1 SQRT-C Building Blocks

The building blocks of the SQRT-C architecture comprises Publisher[1], Subscriber, Monitoring Manager, and clients residing in different locations. We refer to clients as the cloud users who will be hosting their QoS-sensitive applications in the cloud and hence will be interested in obtaining timely resource usage information at the specified QoS levels and intervals of time. Each client consists of command line interface APIs to subscribe to the monitoring data from Cluster Nodes.

[1]We use capitalized Publisher and Subscriber to refer to the publisher and subscriber entities in SQRT-C. All other uses will use smallcase.

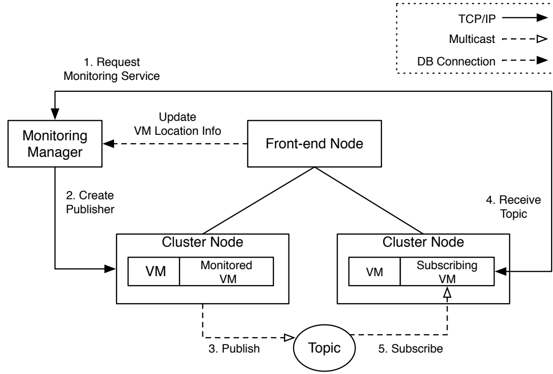

Figure 1: SQRT-C System Architecture

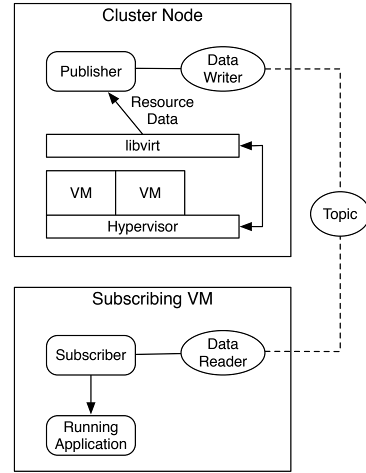

Figure 2: SQRT-C Middleware Communication

Each Cluster Node has a Publisher, which disseminates resource information of virtual machine instances to a Subscriber. A Subscriber is deployed in a client machine (usually a virtual machine) which does auto-scaling and/or provides fault-tolerance for its applications hosted in the cloud. To isolate computation overhead on monitored virtual machines, a Publisher is hosted in a physical Cluster Node and not a deployed virtual machine.

The Monitoring Manager, which is located in the Front-end Node or on an individual physical node (if a database connection is established remotely), serves as an orchestrator to manage DDS connections between Publishers and Subscribers, receiving requests from clients and sending commands to Cluster Nodes.

The remainder of this section describes the need for these building blocks and how they address the design considerations from Section 2.2.

2.3.2 Design Consideration 1: Accessing and disseminating resource information using DDS

Figure 2 depicts the DDS-based communication architecture in SQRT-C to disseminate resource usage information in a timely and scalable manner. Publishers and Subscribers of SQRT-C share common topics, QoS policies, and data structures to accomplish service quality for real-time applications. The contracted variables are agreed upon and controlled to make applications work properly. The Monitoring Manager plays the role of a mediator between DDS publishers deployed in Cluster Nodes and DDS subscribers supplied hosted along with client applications.

Each Cluster Node where Publishers are deployed publishes resource information of selected virtual machines to a Subscriber. The Publisher creates a topic with requested QoS policies by clients and makes a data writer for publishing resource information data of a given virtual machine instance. In order to get resource information of active virtual machines, a Cluster Node first connects to a hypervisor and looks up a virtual machine by an assigned virtual machine ID. If a virtual machine is found, the publisher repeatedly gets the resource information of the virtual machine and disseminates a message containing the resource usage data. The repeating frequency can be defined by users as the latency parameter.

A Subscriber is placed in the same client that manages the infrastructure of the hosted real-time application. It receives resource information of currently running virtual machines to determine whether to increase or decrease resources for its applications. Recall that the QoS of hosted applications should be met to assure correctness of the application, and resource entitlement decision should also be completed within expected deadline to meet service levels of applications. SQRT-C helps meet these objectives by providing QoS-enabled and fine-grained resource information of application infrastructure via Subscribers.

The process works as follows: A client receives a topic from the Monitoring Manager after requesting the monitoring service with virtual machine ID that the client is interested in and specified service levels. A Subscriber creates the received topic and a data reader for receiving resource data of requested virtual machines from a Publisher. The data can be worked with user-defined implementation for managing cloud resources for specific services. The role of the Monitoring Manager is explained next.

2.3.3 Design Consideration 2: Managing DDS Entities through the Monitoring Manager

DDS entities in the cloud are managed by the Monitoring Manager, which serves three purposes. First, it helps in orchestrating the deployment of DDS data-writers and data-readers of the DDS pub/sub mechanism. Second, it ensures that the contract between publishers and subscribers is void of any conflicting QoS policies configured within the DDS mechanisms. Third, since different applications hosted in the cloud may have different requirements on the amount of resource information to be monitored and the time intervals at which it must be monitored, the creation of the right DDS topics and their association with the right data-writers and data-readers is automated by the Monitoring Manager.

The Monitoring Manager manifests under two roles: a server and a client. The Monitoring Manager Server role supervises monitoring services containing status of pub/sub communication between virtual machines, pub/sub topic management, and database connection with OpenNebula. Monitoring Manager has three operations to manage topics (create, terminate, and show). Once the Monitoring Manager Server gets a request from clients for specific monitoring in-

formation and QoS level, it determines where the requested virtual instances are deployed in the cloud and accordingly instantiate a Publisher entity the physical node that hosts the virtual machine.

The Monitoring Manager Client role is required to manage monitoring services that are required to start, terminate, and display the requested services on client side. When a monitoring service is started, a topic is given to the user via the Monitoring Manager Client, and then the user needs to execute a Subscriber to receive monitoring data with the assigned topic as an argument. If the client wants to stop subscribing to the monitoring data, the service can be terminated by executing the terminating command provided by the Monitoring Manager Client.

2.3.4 Design Consideration 3: Shielding Cloud Users from DDS Configurations

SQRT-C provides a standard web-based API for applications to define their resource monitoring requirements thereby shielding them from DDS configurations and deploying DDS publishers and subscribers. To begin using a monitoring service, a client makes a request with information such as interested virtual machine IDs, time delay, and desired QoS as arguments to the Monitoring Manager. Once the Monitoring Manager receives request from a client, the following steps are transparently processed. First the Monitoring Manager interacts with the Front-end Node to obtain information about locations where the virtual machines of interest are physically deployed. Once the request is accepted, a topic is created by the Monitoring Manager. Furthermore, the Monitoring Manager will remotely execute data dissemination by automatically executing Publishers in Cluster Nodes of interest with above created topic and arguments passed by the client. Finally, a Subscriber is also started on the client with same topic enabling it to receive messages published by various Publishers.

3. EXPERIMENTAL RESULTS

This section reports on the experimental results comparing SQRT-C with the contemporary web-based (RESTful) resource monitoring approach (used primarily for virtual resource monitoring). The experiments we conducted aim to prove our hypotheses that SQRT-C outperforms the RESTful monitoring approach in terms of (a) latency, (b) scalability, and (c) jitter.[2] In this paper, we compared the performance with only a RESTful service as it is a conventional communication service used in current cloud platforms, but comparing with other communication middleware technologies such as RMI, and SOAP would be needed to strengthen our hypothesis in the future.

3.1 Testbed

Our cloud infrastructure for the experiments comprises a cluster of 56 blades, 7 rack servers, and Gigabit switches. The servers are operated using OpenNebula to provide hybrid cloud services. Each blade used in our tests contained the following hardware configuration: dual 2.8 GHz Xeon CPUs, 1GB of RAM, 40GB of HDD, and 4 Gigabit Ethernet cards. The following is the hardware specification of a rack server used as a clustered node: 12-core 2.1 GHz Opteron CPUs, 32GB of RAM, 8TB of HDD, and 4 Gigabit Ethernet cards.

OpenSUSE 11.4 Linux was installed as host operating systems for front-end and clustered nodes. Xen 3.0 was installed to provide virtual machines, and our system adopted para-virtualization supporting Ubuntu 11 and CentOS 6 as guest operating systems. OpenNebula 3.0 was used for cloud services. In our OpenNebula system, shared file systems using NFS (Network File System) for distributing virtual machine images are adopted, and virtual network bridges in each computing node for virtual networking were selected. OpenSplice DDS 5.2 is used for the DDS implementation. For the REST implementation, web.py, a web framework for Python, and mimerender, a Python module for RESTful services, are used.

The size of individual published and subscribed message is approximately 160 bytes. Each message contains resource information such as CPU utilization, CPU time, memory utilization and network utilization (number of bytes received and dropped). Content of each message are described in Table 1.

Table 1: Content of each message

Attribute	Description
instanceID	unique instance ID of a virtual machine
cpuUtil	utilization of CPU
cpuTime	the CPU time used in nanoseconds
maxMem	the maximum memory in KBytes allowed
memory	the memory in KBytes used by a VM
numVirtCpu	the number of virtual CPUs for a VM
net_rx_bytes	number of bytes received per sec
net_rx_drops	number of dropped packets per sec
net_rx_packets	number of received packets per sec
net_rx_errors	number of received errors reported
net_tx_bytes	number of bytes transmitted per sec
net_tx_drops	number of dropped packets per sec
net_tx_packets	number of transmitted packets per sec
net_tx_errors	number of transmitted errors reported
block_errors	number of block errors reported
block_rd_bytes	number of bytes read per sec
block_rd_req	number of requests read per sec
block_wr_bytes	number of bytes written per sec
block_wr_req	number of requests written per sec

3.2 Average Message Latency Comparison

In this section we report on the latency comparisons between SQRT-C and RESTful services. Since SQRT-C uses a pub/sub model of communication, which is inherently asynchronous and one way, computing average message latency is tricky. We choose an approach where it is calculated by increasing the number of publishers for a single subscriber. This is because in a pub/sub model of communication, publishers and subscribers are decoupled from each other, and thus increasing the number of subscribers does not affect the latency as a publisher does not care about how many subscribers are interested in what it is publishing. However, the number of messages received by a subscriber can be increased by increasing the number of publishers for that subscriber. Therefore, increasing publishers for a single subscriber will cause increase in latency for messages received by the subscriber. We therefore create a scenario where a single subscriber is subscribing for resource information from

[2]Due to space constraints we did not include additional results.

up to 50 different virtual instances.

In Figure 3, we demonstrate the scalability by showing average message latency (note the logarithmic scale) as a function of increasing number of virtual instances (*i.e.*, increasing the number of SQRT-C Publishers) that a SQRT-C Subscriber is subscribing to. From the figure we see that average message latency increases in almost a linear fashion when the number of Publishers is increased. Initially, when a client is subscribing to 5 virtual instances, the latency is slightly more than 1.6 milliseconds. The latency keeps increasing slightly as we increase the number of Publishers, and for 50 Publishers, the average latency is around 3.3 milliseconds

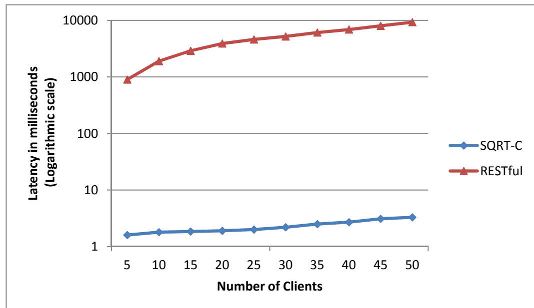

Figure 3: Average Message Latency Comparison of SQRT-C and RESTful by Number of VMs

Unlike SQRT-C, RESTful services use a client-server model of communication which is "pull"-based. In case of SQRT-C, the clients or subscribers, and servers or publishers are completely unaware of each other. However, in case of RESTful service, clients and servers have to be aware of each other since clients request for information from the server on a per-requirement basis. Therefore, to calculate average message latency for RESTful service, we increased the number of clients (subscribers) for a single server (publisher) and measured the round-trip latency

Figure 3 also shows the average message latency as a function of increasing number of clients (subscriber) for RESTful approach. From this figure it is clear that latency increases with increase in number of clients. This is because we have a single server which is serving requests from all of the clients. If we compare this result with the results for SQRT-C average message latency, we can see that both of them show linear increase, however, the latencies observed for RESTful services are orders of magnitude larger than those for SQRT-C. For example, the average latency for RESTful service starts from just less than 1,000 milliseconds (compared to the order of just a few milliseconds in the SQRT-C case) for 5 clients and increases significantly to around 9,200 milliseconds for 50 clients. These results provide an idea of the significant scalability advantage of SQRT-C over RESTful approach.

3.3 Jitter Comparison Between SQRT-C and RESTful Service

To support real-time applications we require our system to provide predictable behavior in terms of message latency. This requires our system to experience minimal jitter since increase in jitter results in unpredictable system. Our hypothesis was that SQRT-C produces significantly less jitter compared to RESTful approach since SQRT-C uses a QoS-enabled pub/sub model of communication. Using different QoS policies, we can make sure that SQRT-C produces very little jitter.

For the purposes of this experiment, we compare message round-trip time of RESTful services and message latency of SQRT-C. The reason behind this is the fact that round trip time cannot be measured in a trivial way for SQRT-C since it uses pub/sub communication model. The decoupled nature of publishers and subscribers in a pub/sub model makes it difficult and to some degree illogical to calculate round trip time for SQRT-C.

Figure 4 demonstrates message response times measured for each message transfer for a total of 500 messages. From Figure 4 we can clearly see that RESTful services produce significant jitter and hence is very unpredictable in

delivering bounded latencies. The message response time is never stable and is always fluctuating between 300 milliseconds and 420 milliseconds.

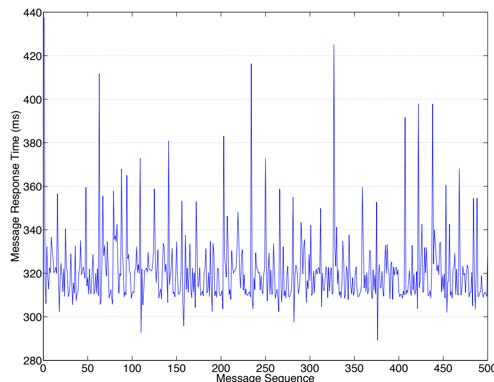

Figure 4: Message Round-trip Time and Jitter of RESTful Service

If we use proper QoS policies for SQRT-C (by configuring the underlying DDS transport), we will be able achieve very stable message latencies. Figure 5 demonstrates message latency over number of messages exchanged. From Figure 5, we can clearly see that using Best Effort reliability results in extremely stable message latency. When using Best Effort reliability, the message latency is always around 2 milliseconds. The reliability setting introduces some jitter and also increases the latencies due to the reliability logic in DDS incurring the bulk of the performance overhead. Given the stable networks in a data center, Best Effort reliability configuration option is acceptable.

4. CONCLUDING REMARKS AND FUTURE WORK

This paper presented a scalable cloud resource monitoring system called SQRT-C that leverages DDS and supports QoS. It is observed through experimental results that DDS in the cloud as a monitoring service is more appropriate for hosting real-time applications that need fine-grained auto-scaling decisions than widely used technologies like RESTful services. It is challenging to use DDS for a cloud monitor-

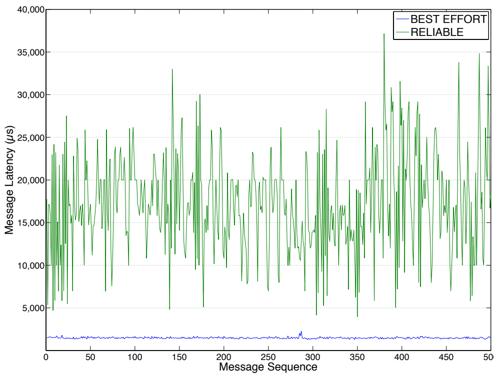

Figure 5: Message Latency of Different Reliability QoS

ing service because in the cloud it is preferred that resource information be obtained as a service rather than directly by accessing physical machines or virtual machines by clients. Also, proper configurations of DDS services according to service levels are not trivial to be defined by clients. The Monitoring Manager in SQRT-C is hence indispensable as a public access point and an orchestrator to furnish services appropriately and automate most of the activities. The use of libvirt library makes it seamless for SQRT-C to be deployed in cloud platforms without any invasive changes to the cloud software.

Future Work: Our future work will comprise the following dimensions of work in this area.

• We did not experiment with different QoS settings of DDS in our preliminary work. Our future evaluations will therefore determine the impact of different DDS QoS settings on the dissemination of information in the cloud.

• Our present work was conducted on the OpenNebula platform only. Although we observed that no invasive changes were needed to integrate DDS, we need to test our approach in a variety of cloud platforms to ascertain this fact.

• A fault-tolerant orchestrator is needed in the cloud in the case of recovering failed virtual machines. SQRT-C will be operated with the fault-tolerant orchestrator and will be more precisely analyzed in practical settings with real applications hosted in the cloud.

• Our experiments did not consider finding the sweet spot when having too many Publishers in a Cluster Node may start impacting overall performance of the applications running in the virtual machines. But this depends on how many virtual machines are hosted in a Cluster Node. Thus, our future work will explore finding the optimal number of virtual machines and Publishers that can be hosted in any Cluster Node such that performance continues to be at acceptable levels while physical resources are utilized maximally.

The source code for SQRT-C is available for download at `www.dre.vanderbilt.edu/~kyoungho/SQRT-C`.

5. REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. 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] T. M. Takai, "Cloud Computing Strategy," Department of Defense Office of the Chief Information Officer, Tech. Rep., Jul. 2012. [Online]. Available: http://www.defense.gov/news/DoDCloudComputingStrategy.pdf

[3] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "The Resource-as-a-Service (RaaS) Cloud," in *USENIX Conference on Hot Topics in Cloud Computing (HotCloud).* USENIX, 2012.

[4] M. Massie, B. Chun, and D. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[5] W. Barth, *Nagios: System and network monitoring.* No Starch Pr, 2008.

[6] C. Huang, P. Hobson, G. Taylor, and P. Kyberd, "A study of publish/subscribe systems for real-time grid monitoring," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International.* IEEE, 2007, pp. 1–8.

[7] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE'08.* Ieee, 2008, pp. 1–10.

[8] F. Han, J. Peng, W. Zhang, Q. Li, J. Li, Q. Jiang, and Q. Yuan, "Virtual resource monitoring in cloud computing," *Journal of Shanghai University (English Edition)*, vol. 15, no. 5, pp. 381–385, 2011.

[9] S. De Chaves, R. Uriarte, and C. Westphall, "Toward an architecture for monitoring private clouds," *Communications Magazine, IEEE*, vol. 49, no. 12, pp. 130–137, 2011.

[10] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Internet of Things (IOT), 2010.* IEEE, 2010, pp. 1–8.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computer Survey*, vol. 35, pp. 114–131, June 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[12] *Data Distribution Service for Real-time Systems Specification*, 1.2 ed., Object Management Group, Jan. 2007.

[13] A. Corsaro, "10 reasons for choosing opensplice dds," 2009. [Online]. Available: http://www.slideshare.net/Angelo.Corsaro/10-reasons-for-choosing-opensplice-dds

[14] D. Schmidt and H. van't Hag, "Addressing the challenges of mission-critical information management in next-generation net-centric pub/sub systems with opensplice dds," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE, 2008, pp. 1–8.

[15] A. Corsaro, L. Querzoni, S. Scipioni, S. Piergiovanni, and A. Virgillito, "Quality of service in publish/subscribe middleware," *Global Data Management*, pp. 1–19, 2006.

[16] J. Fontán, T. Vázquez, L. Gonzalez, R. Montero, and I. Llorente, "Opennebula: The open source virtual machine manager for cluster computing," in *Open Source Grid and Cluster Software Conference*, 2008.