2011 Fourth IEEE International Conference on Utility and Cloud Computing

An Extensible Cloud Platform Inspired by Operating Systems

Akiyoshi Sugiki and Kazuhiko Kato
Department of Computer Science, University of Tsukuba
Tsukuba, Japan
{sugiki, kato}@cs.tsukuba.ac.jp

***Abstract*—Virtualization has changed the ways computation is done, especially in utility and cloud computing. While several virtualization cloud platforms have emerged to provide interfaces between users and data centers, traditional operating systems (OSes) have defined interfaces between applications and hardware. We argue that virtualization cloud platforms and OSes can share many problems; thus, many techniques developed for OSes can be applied to such platforms. We have implemented a prototype of an OS-like middleware that adopted a microkernel design, had resource abstraction achieved by objects and functions, and offered a scripting environment as a programming interface. The experimental results demonstrated that it had the ability to achieve elasticity and high-availability VMs. We also carried out qualitative comparisons with other middleware to clarify our research position.**

***Keywords*-cloud computing, IaaS; middleware, virtualization**

I. Introduction

Utility and cloud computing have changed the shape of computing. This is bringing about a large shift in computer resources away from client sites into federations of data centers. Since virtualization is playing an important role in such computing and its advantages of resource multiplexing, isolation, and flexibility perfectly fit into a class of Infrastructure-as-a-Service (IaaS) clouds, many virtualization cloud platforms such as Eucalyptus [1], OpenNebula [2], Nimbus [3], Xen Cloud Platform [4], and OpenStack [5] have emerged.

The role of virtualization cloud platforms is to manage various types of resources, including virtual machines (VMs) and physical machines, networks, and storage in data centers. However, traditional operating systems (OSes) have emerged to manage various hardware, such as CPUs, memories, disks, and other input/output devices. Thus, both goals are quite similar from the broad perspective of providing superior resource abstraction, attaining high resource efficiency, as well as providing user interfaces. Although they certainly have differences such as distribution and resources that are actually managed, designers of virtualization platforms can learn many things from the long history of OSes.

This paper presents a virtualization cloud platform inspired by OSes, especially from two streams: Unix and microkernels. As Unix has done, we first clearly separated middleware functionality into a *shell* and *kernel*. We next placed a minimized core for communications in the kernel at the bottom. Then, resources were abstracted as *distributed objects* and manipulated with *parallel functions*. Finally, our use of a shell introduced flexibility and customizability. We offered a dual stack of Application Program Interfaces (APIs) in the shell; the first was a direct API referred to as "Kumoi" to manipulate data centers directly and second was a cloud API called "Kali" to build a complete stack for cloud computing.

The purpose of this study was not to build another competitor for production-quality platforms; rather, we were interested in exploring the design space of such platforms. Our system was mainly designed for researchers. That is, many researchers can verify new techniques, such as energy-aware scheduling, fault-tolerances, and autonomic computing using our platform within short periods of time. We also intended for developers to create highly customized clouds according to their specific needs in underlying hardware and service-level agreements (SLAs).

Although this paper is an enhancement of the previous one [6], we have made several non-trivial contributions. The previous system was never discussed from the aspect of OSes – It was just a scripting tool with no clear separation of the shell and kernel. There was also no support for requisite features of cloud computing, such as resource pools, elasticity, and high-availability.

Our research is still at an early stage but the preliminary experiments demonstrated the system's ability to achieve elasticity and high-availability VMs. Finally, a qualitative comparison with other platforms was demonstrated to clarify our position.

The rest of the paper is organized as follows. We first discuss our research motivation in Section II. Section III provides a design overview and Section IV describes the system's implementation. We explain how users interacted with our scripting environment in Section V and discuss an evaluation using example scripts in Section VI. We survey related work in Section VII and conclude the paper in Section VIII with a summary of the key points.

II. Motivation

This section discusses several key insights that directed us to OS-like middleware. We analyzed virtualization cloud

978-0-7695-4592-9/11 $26.00 © 2011 IEEE
DOI 10.1109/UCC.2011.49

platforms and found OSes share five main problems:

Resource Abstraction: One of the main OS tasks is to manage various kinds of resources. Many OSes provide resource abstractions to achieve this end. For example, Unix has offered a file abstraction that was later extended in Plan9 [7], and more recent OSes have abstracted resources as objects. However, the role of a cloud platform is quite similar in that it has to manage various resources such as VMs, physical machines, networks, and storage. Although they often provide APIs, the use of a different API for each resource is required. Thus, providing uniform access to various kinds of resources is helpful. Such abstraction has been repeatedly explored in OSes; however, we applied it to the cloud platform context.

Execution and Scheduling: Another OS task is to execute programs along with data. CPU schedulers have especially been the state-of-the-art part of many OSes. Their role in a cloud platform has changed to execute VMs. Despite the difference in scheduling targets, both OS and cloud platform schedulers are interested in how efficiently resources are managed to satisfy all resource demands.

As previously described, there are many similarities between OSes and virtualization cloud platforms, but they also have the following differences:

Resource Granularity: Obviously, the biggest difference is the granularity of resource control. While traditional OSes manage resources at finer granularities such as processes or memory regions, virtualization cloud platforms manage resources at coarser VM-levels on the top of traditional OSes. As VM migration has simplified problems in classical research on process migration [8], many other problems in distributed OSes should also be simplified.

Communication Cost: Most criticism of early microkernels in OSes such as Mach [9] was directed at the communication overhead. Thus, much subsequent research such as that by L4 [10] and Exokernel [11] had to focus on improving its performance. Such overheads, on the other hand, will be amortized in virtualization cloud platforms because VM booting and communication between distant machines takes a long time. Thus, virtualization middleware can adopt many techniques once rejected within the OS context due to overheads.

Scalability: However, virtualization middleware poses another challenge, i.e., scalability. Researchers of distributed OSes once explored design space in distributed environments, but their scalability was only limited to tens or hundreds of workstations. Thus, their techniques could not be applied to cloud environments where thousands of machines are installed.

In addition, we have added another aspect of OSes to virtualization middleware:

Scripting: Primary shell support might be the most prominent of the many contributions made by Unix. The Unix shell allows users to do small tasks without duties and rapidly create prototypes. It also offers customizability that allows users to tailor the environment. These advantages are not only derived from the shell language itself, but also from many well-designed commands and several powerful abstractions such as pipes and redirections. We expect this "sum of the parts being greater than the whole" [12] will also be helpful in virtualization middleware.

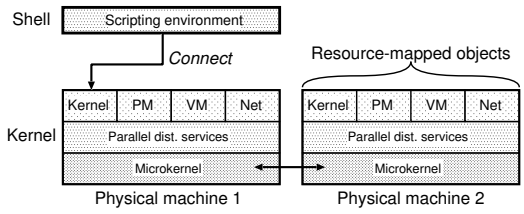

Figure 1. Kumoi/Kali Architecture

III. System Design

A. Overview

Figure 1 outlines our software architecture. A prototype called Kumoi/Kali was implemented in Scala [13] and has now reached about 27-K lines of code. The system consists of four components:

Microkernel: A minimum kernel approach was taken, which is much like that in OSes. This layer is responsible for communications so that all access to remote resources gets through this layer. Many other functions were implemented as objects or services, which will be explained later.

Resource-Mapped Objects: We applied resource abstraction by using an object-oriented paradigm to provide a uniform view of various resources. Resource-mapped objects provided bi-directional mapping between actual resources and corresponding language objects. This approach also enabled remote access using distributed object techniques implemented in the microkernel.

Parallel Distributed Services: We provided parallel distributed functions that could be applied to a collection of the above objects. This parallel execution service was useful for increasing scalability. We also provided a membership service that automatically manages nodes in a cluster.

Shell: The shell introduced flexibility and customizability over a platform. Most cloud platform features were implemented by scripting.

Kumoi/Kali was clearly separated into kernel and shell parts. A kernel must be installed on each node at a data center. Such kernel nodes were managed in a *decentralized manner* where there was no single point of central management. However, a shell could run on the administrator side. An administrator could issue any operations via relaying that machine the shell that it was connecting to because of decentralized management.

The rest of this section describes how each kernel component was implemented. The shell interface is also described in Section V.

B. Microkernel

Actually, our microkernel was implemented using Java RMI and Scala Actors. Because the resources in our architecture were abstracted as objects, Java RMI was used to access remote objects. Scala Actors, on the other hand, were used for asynchronous communications such as event notifications similar to those by Unix signals. They were also used to implement the distributed algorithms in the internals. Despite its original concept, we actually used the actors like the light-weight processes in OSes.

One of the advantages of this architecture was that our microkernel could be a single mediation point for all access to resources because their access must get through this layer. Therefore, we could incorporate a security mechanism or fault-tolerant mechanism in future work by extending this layer.

Our microkernel architecture was initially successful but there is much room for improvement. First, our microkernel could be made smaller because the current microkernel ran on top of the Java VM on a traditional OS, along with a large number of Java standard libraries. Although such an architecture greatly simplified development, another approach is possible. Second, although we mixed Java RMI and Scala Actors, there might be a slight gap in the architecture. This is because the actors were higher abstractions that hid many underlying details such as address space, threads, and schedulers. Although we are seeking a more suitable architecture, they worked quite well without corrupting the design.

C. Resource-Mapped Objects

As previously described, we encapsulated data center resources as language objects. We called such objects resource-mapped objects (RMOs). Although this approach might sound similar to implementing virtualization middleware using an object-oriented language, there are large differences.

First, one of these differences arose because we made the lifetimes of RMOs directly correspond to those of resources. That is, we could create and destroy a VM resource by creating and deleting a corresponding language object, for example. Table I lists the methods of managing the lifecycles of RMOs. We borrowed the idea from the create, read, update, and delete (CRUD) semantics for Web frameworks, even though additional methods such as add, index, and clone were supplemented from the original.

The second difference arose because RMOs also provide a uniform set of methods for different kinds of resources. These carefully designed interfaces could be used for many resources such as VMs, physical machines, and networks.

Table I
CRUD SEMANTICS IN RMOS

Operation	Example code
Create	`val v = vmm.createVM`
Add	`vmm.add(v)`
Read	`val n = v.name`
Update	`v.name = "Fedora 15"`
Delete	`vmm.remove(v)`
Index	`val vlist = vmm.vms`
Clone	`val c = v.clone`

Table II
COMMON RMO METHODS

Method	Description
`name`	Get resource name
`uuid`	Get resource unique ID
`info`	Get resource static information
`stats`	Get resource statistics
`watch(actor)`	Set monitor actors
`unwatch(actor)`	Unset monitor actor
`dump`	Dump resource information
`patch(diff)`	Patch object
`acls`	Get access control lists
`account`	Get accounting information
`audit`	Get audit logs

Table II summarizes common methods in various resources. Many methods are available to acquire information and provide feedback.

Our approach of using RMOs had several advantages. First, it was easy to handle them in a scripting environment because actual resources could be manipulated as language objects. Second, it could take advantage of many techniques once it had been developed for the object-oriented paradigm. For example, remote access was quite naturally implemented using distributed object techniques such as Java RMI. Thus, our approach could seamlessly integrate classical techniques within the brand-new cloud computing context.

D. Parallel Distributed Services

Although our microkernel and RMOs were sufficient to manage a data center, we also provided parallel distributed functions to increase scalability. Below, is an example of our parallel distributed functions.

```
pms.dmap(_.name)
```

The `dmap` function is actually a parallel distributed function that was applied to the `pms`, i.e., a list of available physical machine RMOs. The `dmap` looks like a standard `map` function of Scala, but it was calculated in a parallel and distributed way. We also provided many other functions such as `dfilter` and `dreduce`.

Actually, these functions were implemented using a similar technique to the parallel collections of Scala 2.9, but our approach was extended to utilize multiple physical machines. A custom class loader was incorporated to remotely transfer functions passed as arguments. The original parallel

collection was also used to efficiently utilize available cores on each machine.

Another parallel distributed service is a membership one. Our membership service automatically maintained a list of available machines. This list was supplied as the `pms`, i.e., a list of physical machine RMOs, to the shell. A variant of the well known Gossip protocol [14] was used in the current implementation.

IV. Implementation

This section briefly describes how RMOs were implemented. The range of supported resources could easily be extended by adding other RMOs. The current implementation supports Xen, KVM, and QEMU by using Libvirt for the underlying implementation. Virtual networking is also supported by using Open vSwitch.

Because security is one of the major issues in cloud computing, our system provided a framework for that purpose. It was implemented using the Java Authentication and Authorization Service (JAAS) and Java Sandbox. This compels users to be authenticated by a Lightweight Directory Access Protocol (LDAP) server and restricts them in a sandbox.

V. Application Interface

The shell interface was also implemented in Scala. A shell user could use this interactively in the Scala's repeat-eval-print-loop (REPL) environment or as part of scripting in a Scala program.

We provided two layers of APIs as a shell interface. The first was a lower set of APIs for directly manipulating the physical structure of a data center. The second was a richer set of APIs for building a cloud platform. A shell user could choose either API depending on the purpose.

A. Direct API

As previously mentioned, a direct API provided ways of manipulating the structure of a data center. This API is quite useful for system researchers who are investigating techniques in a cloud platform such as energy-aware scheduling, fault-tolerance, and autonomic computing. This API is also useful for administrators in maintaining the health of data centers.

VM Contextualization: One powerful example of direct API is VM contextualization. Although VM contextualization is also possible in several middleware such as Nimbus [3] and OpenNebula [2], it could be achieved quite seamlessly in our environment.

The following example created 100 VMs using a context. Because we integrated configuration file management into the shell, most administrative tasks could be achieved without leaving the shell environment.

```
val context = (no: Int) => {
  val v = pms(no).vmm.createVM
  v.name = "centos" + no
  v.add(FileDiskAuto("/nfs/centos" + no + ".img"))
  v.add(Bridge("xenbr", "vnet" + no, mac|no))
  pms(no).vmm.add(v)
}
(0 until 100).map(context(_))
```

VM Monitoring: Because the RMOs already provided the methods of monitoring and providing feedback, a VM monitoring task could easily be implemented. The following code distilled VMs whose CPU ratio was greater than 90%.

```
pms.dmap(_.vms.filter(_.cpuRatio > 0.9)).flatten
```

B. Kali: Cloud API Prototype

Although direct API provided a powerful environment for manipulating a data center, a gap still remained between the API that was provided and the one that was required to build a cloud. Because direct API did not provide any features that satisfied the NIST cloud definition [15] such as resource pools, on-demand self-services, or rapid elasticity, users of direct API will have to write additional amounts of code by scripting.

Our cloud library prototype called Kali could be used for such purposes. Figure 2 shows an example of use, where a resource pool was created with the computing resources of several physical machines, and a single VM was added with elastic capabilities.

The following describes all related components:

Global object: The global object is a special object that is unique in a data center. All data-center-wide features, such as resource pools and the security mechanism, are referenced from that object. It was currently implemented as a single object in the global node, but we are now developing one that is more complex whose state is replicated on multiple nodes based on the Paxos protocol [16] or ZooKeeper [17]. Even if such changes are incorporated, we do not have to change the global object interface.

Resource pools: A resource pool RMO manages a collection of physical machines as pooled computing resources. Its task is to assign the computing resources to VMs according to scheduler decisions.

Scheduler: A scheduler RMO actually determines the placement of VMs on the physical machines in a resource pool. Figure 3 shows an example of the internal implementation of such schedulers. This is a simplified version of the well-known rank scheduler used in OpenNebula. This example reveals how concisely such a scheduling algorithm can be written in our scripting environment.

Elasticity and high-availability wrappers: Elasticity and high-availability VM wrappers are also implemented as RMOs. Elasticity is achieved by using an elastic scheduler and copy-on-write (CoW) VM image features. An elastic scheduler basically works for the user side and it spawns VM-clones if more capacity is required. Similarly, high-availability is achieved by monitoring the health of a VM and rebooting it if it disappears.

```
val p = global.createPool
p.name = "pool1"
global.add(p)

val pool = global.pools(0)
pool.add(pms.filter(_.name.startsWith("hpc")))
pool.add(pool.createElasticity(v))
```

Figure 2. Example of use of cloud API

```
def schedule(v: ColdVM,
  pms: List[HotPhysicalMachine]) = {
  val candidates = pms.filter(_.availableFor(v))
  val sorted = candidates.sortWith(fsort)
  val selected = sorted.head
  selected.vmm.add(v)
}
```

Figure 3. Simplified rank scheduler

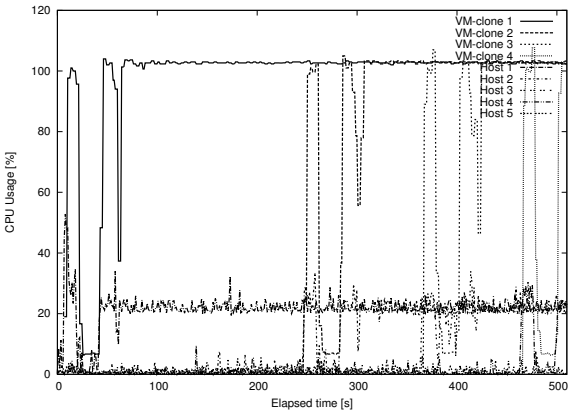

Figure 4. Elasticity of VM

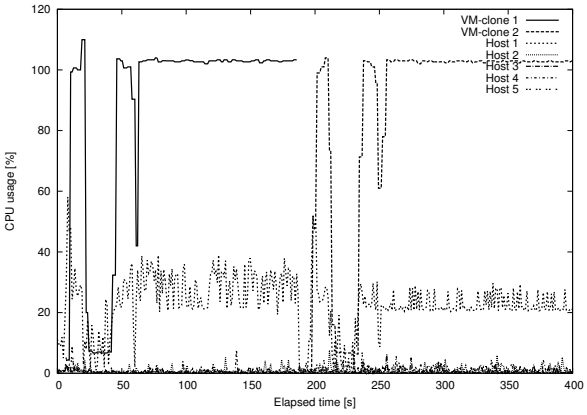

Figure 5. High availability of VM

VI. Preliminary Experiments

This section briefly discusses our evaluation of the system in terms of functionality. It was assessed by demonstrating elasticity and high-availability VMs.

A. Experimental Setup

The experiments were conducted on a 5-node cluster with a quad-core Xeon L5410 CPU that had 8 GB of memory and a single 320-GB ATA drive. The platform was run using Fedora 14 (Linux 2.6.35.x86_64), KVM 0.13, Libvirt 0.8.6, and Scala 2.9.0final. A single network file system (NFS) server with dual Xeon E5620 CPUs, 24 GB of memory, and 1.8-TB RAID5 drives was used to accomplish live migration of VMs. These servers were interconnected by a 1000BASE-T network.

B. Elasticity and high-availability of VMs

Figure 4 shows the CPU ratios as they changed over time. These ratios were measured by writing a monitoring script in Kumoi. Our test wrapper for VM elasticity monitored the average CPU usage of VM clone instances and spawned another VM clone if the average ratio exceeded a threshold. The VM used in the experiment artificially consumed CPU time after the OS was booted to demonstrate elasticity.

The VM clone in Figure 4 initially ran on Host 1. Our VM wrapper detected the violation of a threshold and spawned a VM clone in 246 sec. Other clones of the VM were also spawned on Host 3 in 364 sec and on Host 4 in 463 sec.

Figure 5 shows the results with a high-availability feature. Our example VM wrapper periodically monitored the availability of a VM and rebooted it if it was unavailable. The VM clone in Figure 5 initially ran on Host 1 and was artificially destroyed in 180 sec. Our VM wrapper detected such unavailability and rebooted another clone of the VM in 197 sec.

VII. Related Work

Many virtualization cloud platforms such as Eucalyptus [1], OpenNebula [2], Nimbus [3], Xen Cloud Platform (XCP) [4], and OpenStack [5] have recently emerged. Although several papers that have compared them [18], [19], [20] are available, unfortunately, none of these have discussed quantitative evaluations of such platforms. This is because these platforms differ quite markedly in their architecture, and it is thus difficult to find fair criteria to compare their performance.

Therefore, we have only discussed qualitative comparisons of Kumoi/Kali with other platforms in this paper. The results are summarized in Table III. Obviously, we took the different approach of using a microkernel with RMOs although most of the others had well-designed structures attained by using components. We enhanced such component-based architecture by adopting minimized-core and resource-mapping techniques. Although Kumoi/Kali does not yet provide a complete stack of functionality and reliability compared to the others, it could easily be improved by using the extensibility of our platform. Kumoi also provided low-level access via direct API whereas most of the others were made possible with source-code modifications. Although such modifications were not very hard to achieve

Table III
QUALITATIVE COMPARISON WITH SEVERAL CLOUD PLATFORMS

Platform	Architecture	Functionality	Low-level Access	Command-line I/F	Reliability	Customizability	External API
Eucalyptus	Components	Good	Possible	Unix shells	Good	Possible	EC2-compatible API
OpenNebula	Components, Modular	Good	Possible	Unix shells	Good	Possible	EC2 and OCCI APIs
Nimbus	Components	Good	Possible	Unix Shells	Good	Possible	WSRF and EC2 APIs
OpenStack	Components	Good	Possible	Unix Shells	Good	Possible	OpenStack and EC2 APIs
XCP	Language API + CLI	Good	Good (via XAPI)	Unix Shells	Good	Good	XCP API
Kumoi/Kali	Microkernel + RMOs	In Progress	Good	Language-integrated	Improvable	Very-good	Kumoi/Kali API

by using sophisticated internals, they become easier in our environment. Our command-line interface was integrated into a language while most of the others provided Unix shell-based commands. Although we have not currently provided an external interface directly, such as an Amazon EC2-compatible API, this could easily be implemented by using our RMI and actor-based APIs.

Although the purpose of this paper was to present the design of an OS-like cloud platform, we are not the only ones who have advocated the need for a data center OS. Zaharia et al. [21] also suggested this need within a slightly different context. They focused on large-scale data processing, which is typically represented by MapReduce and Dryad/LINQ, and they suggested fine-grain resource sharing by the Mesos OS [22]. Although their research and ours focused on different targets, data processing, and IaaS, they are in a complementary relationship. They basically focused on resource scheduling whereas we focused on providing resource abstraction.

Apart from OSes, many distributed system techniques such as Java RMI and sandbox could also be applied to the cloud context. We explored the possibility of seamlessly integrating classical techniques and a brand-new context on this platform.

VIII. CONCLUSION

This paper has presented a virtualization cloud platform inspired by OSes. We argued that both OS and cloud platform roles were quite similar in that they were designed to provide resource abstractions over hardware and simplify the development of applications. We plan a variety of extensions in future work, including fault-tolerance and security mechanisms, by replacing existing components. Such incremental extensibility would be enabled by our powerful abstractions that were inspired by OSes.

ACKNOWLEDGMENTS

This work was supported in part by a Japan SCOPE grant, 092003021, and KAKENHI grants 2230006 and 22700023.

REFERENCES

[1] D. Nurmi *et al.*, "The Eucalyptus Open-Source Cloud-Computing System," in *IEEE/ACM CCGrid'09*, 2009, pp. 18–21.

[2] OpenNebula Project Leads, "OpenNebula: The Open Source Toolkit for Cloud Computing," 2008, http://www.opennebula.org/.

[3] P. Marshall, K. Keahey, and T. Freeman, "Elastic Site: Using Clouds to Elastically Extend Site Resources," in *IEEE/ACM CCGrid 2010*, 2010, pp. 43–52.

[4] Citrix, "Xen Cloud Platform – Advanced Virtualzation Infrastructure for the Clouds," 2009, http://www.xen.org/products/cloudxen.html.

[5] Rackspace Cloud Computing, "OpenStack: The Open Source, Open Standards Cloud," 2010, http://openstack.org/.

[6] A. Sugiki *et al.*, "Kumoi: A High-Level Scripting Environment for Collective Virtual Machines," in *IEEE ICPADS 2010*, 2010, pp. 322–329.

[7] R. Pike *et al.*, "Plan9 from Bell Labs," *Computing Systems*, vol. 8, no. 3, pp. 72–76, 1995.

[8] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, pp. 241–299, September 2000.

[9] M. Accetta *et al.*, "Mach: A New Kernel Foundation for UNIX Development," in *Summer 1986 USENIX Conf.*, 1986, pp. 93–112.

[10] H. Härtig *et al.*, "The Performance of μ-kernel-based Systems," in *ACM SOSP'97*, 1997, pp. 66–77.

[11] M. F. Kaashoek *et al.*, "Application Performance and Flexibility on Exokernel Systems," in *ACM SOSP'97*, 1997, pp. 52–65.

[12] M. Gancarz, *The UNIX Philosophy*. Butterworth-Heinemann, 1996.

[13] M. Odersky, "The Scala Programming Language," 2003, http://www.scala-lang.org/.

[14] R. van Renesse, Y. Minsky, and M. Hayden, "A Gossip-style Failure Detection Service," in *IFIP Middleware'98*, 1998, pp. 55–70.

[15] National Institute of Standards and Technology, "NIST Definition of Cloud Computing," 2009, http://www.nist.gov/itl/cloud/.

[16] L. Lamport, "The Part-time Parliament," *ACM TOCS*, vol. 16, no. 2, pp. 133–169, 1998.

[17] P. Hunt *et al.*, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX Annual Tech. Conf.'10*, 2010, pp. 11–11.

[18] T. D. Cordeiro *et al.*, "Open Source Cloud Computing Platforms," in *Int'l Conf. on Grid and Cloud Computing*, 2010, pp. 366–371.

[19] J. Peng *et al.*, "Comparison of Several Cloud Computing Platforms," in *Int'l Symp. on Information Science and Engieering*, 2009, pp. 23–27.

[20] P. Sempolinski and D. Thain, "A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus," in *IEEE CloudCom 2010*, 2010, pp. 417–426.

[21] M. Zaharia *et al.*, "The Datacenter Needs an Operating System," in *USENIX HotCloud 2010*, 2010.

[22] B. Hindman *et al.*, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *USENIX NSDI 2011*, 2011.