Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds

Eugen Feller, Louis Rilling, Christine Morin

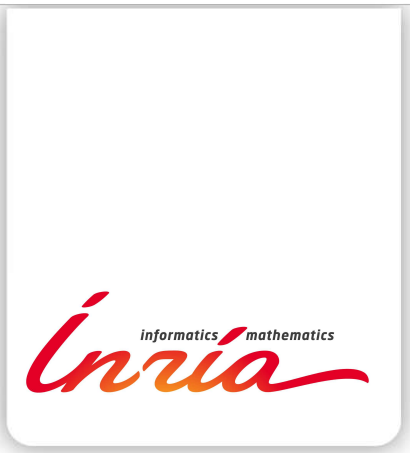
RESEARCH REPORT

N° 7833

December 2011

Project-Teams MYRIADS

ISSN 0249-6399 ISRN INRIA/RR--7833--FR+ENG



Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds

Eugen Feller*, Louis Rilling†, Christine Morin *

Project-Teams MYRIADS

Research Report n° 7833 — December 2011 — 20 pages

Abstract: With the advent of cloud computing and the need to satisfy growing customers resource demands, cloud providers now operate increasing amounts of large data centers. In order to ease the creation of private clouds, several open-source Infrastructure-as-a-Service (IaaS) cloud management frameworks (e.g., OpenNebula, Nimbus, Eucalyptus, OpenStack) have been proposed. However, all these systems are either highly centralized or have limited fault-tolerance support. Consequently, they all share common drawbacks: *scalability is limited by a single master node* and *Single Point of Failure (SPOF)*.
In this paper, we present the design, implementation and evaluation of a novel scalable and fault-tolerant virtual machine (VM) management framework called *Snooze*. For scalability our system utilizes a self-organizing hierarchical architecture and performs distributed VM management. Moreover, fault-tolerance is provided at all levels of the hierarchy, thus allowing the system to self-heal in case of failures. Our evaluation conducted on 144 physical machines of the Grid'5000 experimental testbed shows that the fault-tolerance features of the framework do not impact application performance. Moreover, negligible cost is involved in performing distributed VM management and the system remains highly scalable with increasing amounts of resources.

Key-words: Cloud computing, fault tolerance, scalability, self-organization, self-healing, virtualization

* INRIA Centre Rennes - Bretagne Atlantique, Campus universitaire de Beaulieu, 35042 Rennes, France - {Eugen Feller, Christine Morin}@inria.fr

† Kerlabs, 80 avenue des buttes de Coësmes, Bâtiment Germanium, 35700 Rennes, France - l.rilling@av7.net

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Snooze : un gestionnaire de machines virtuelles pour nuages informatiques privés autonome et passant à l'échelle

Résumé : Avec l'émergence de l'informatique en nuage et l'augmentation des demandes en ressources par les utilisateurs, les fournisseurs de nuages informatiques utilisent de plus en plus de centres de données de grande taille. Afin de faciliter la création de nuages informatiques privés, plusieurs gestionnaires de nuages informatiques de type Infrastructure-as-a-Service (IaaS) ont été proposés sous forme de logiciel libre (e.g., OpenNebula, Nimbus, Eucalyptus, OpenStack). Cependant, tous ces systèmes sont fortement centralisés ou ont un support limité pour la tolérance aux pannes. Par conséquent, ils partagent tous les mêmes limitations : passage à l'échelle limité par un nœud maître unique et point unique de défaillance (SPOF).

Dans cet article, nous présentons l'architecture, la mise en œuvre et l'évaluation d'un gestionnaire de machines virtuelles novateur, passant à l'échelle et tolérant aux pannes, nommé Snooze. Pour passer à l'échelle, notre système utilise une architecture hiérarchique auto-organisante et effectue une gestion distribuée des machines virtuelles. De plus, la tolérance aux pannes est fournie à tous les niveaux de la hiérarchie, permettant ainsi au système de s'auto-réparer en cas de panne. Notre évaluation réalisée sur 144 machines physiques de la plateforme expérimentale Grid'5000 montre que la tolérance aux pannes de notre gestionnaire n'influe pas sur les performances des applications. De plus, le coût de la gestion distribuée des machines virtuelles est négligeable, permettant à notre système de passer à l'échelle pour un nombre croissant de ressources.

Mots-clés : Informatique en nuage, tolérance aux pannes, passage à l'échelle, auto-organisation, auto-réparation, virtualisation

1 Introduction

Cloud computing has recently appeared as a new computing paradigm which promises virtually unlimited resources. Customers rent resources based on the pay-as-you-go model and thus are charged only for what they use. In order to meet growing customers resource demands, public IaaS-cloud providers (e.g., Amazon EC2, Rackspace) are now operating increasing numbers of large data centers.

Several open-source IaaS-cloud management frameworks such as OpenNebula [18], Nimbus [14], Eucalyptus [20], and OpenStack [5] have been developed in order to facilitate the creation of private clouds. Given the ever growing computing power demands, they need to scale with increasing number of resources and continue their operation despite system component failures. However, all these frameworks have a high degree of centralization and do not tolerate system component failures. For example, the scalability of OpenNebula, Nimbus and OpenStack is limited by their frontend node/cloud controller. Moreover, as no replication support exists, a failure of the frontend node makes the VM management impossible. Similarly, in Eucalyptus each non fault-tolerant cluster controller suffers from the same drawbacks. Last but not least, no VM monitoring is performed thus limiting the support for advanced VM placement policies (e.g., consolidation).

In order to provide scalability and fault-tolerance to virtualized data centers, we proposed Snooze [12], a *novel VM management framework for private clouds* which is designed to scale across thousands of nodes. Unlike the existing cloud management frameworks, Snooze utilizes a *self-organizing hierarchical architecture* and performs *distributed VM management*. Particularly, VM management tasks are performed by multiple managers, with each manager having only a *partial* view of the system. Moreover, fault-tolerance is provided at all levels of the hierarchy by replication and integrated *leader election algorithm*. Consequently, the system is able to self-heal and continue its operation despite system component failures. Finally, VM monitoring is integrated into the framework and a generic scheduling engine exists to support advanced scheduling policies.

The contribution of this paper is as follows. We give an *update on the design of Snooze* and now present its *implementation specific details*. Snooze is fully implemented from scratch in Java and currently comprises approximately *15,000 lines of code*. Moreover, the *framework evaluation* is presented. Snooze was evaluated by deploying it on 144 machines of the Grid'5000 experimental testbed [9] and submitting up to 500 VMs. Three aspects were studied: VM submission time in a centralized as well as distributed deployment, overhead of fault-tolerance on application performance and scalability aspects. Note, that the evaluation of different scheduling policies is out of the scope of this work. The results show that submission time is not impacted by performing distributed VM management. Moreover, the proposed system remains highly scalable with increasing number of resources and its self-organization properties do not impact application performance.

The remainder of this paper is organized as follows. Section 2 introduces the design and implementation of Snooze. Section 3 presents the evaluation of the framework. Section 4 discusses related work. Finally, Section 5 closes the paper with conclusions and future work.

2 Design and Implementation

This section describes the architecture and implementation of Snooze. First, the system model is presented. Afterwards, a global overview of the architecture is given and the system components are presented. Finally, the self-organization and self-healing mechanisms are described.

2.1 System Model and Assumptions

We assume a data center whose nodes are interconnected with a high-speed LAN connection such as Gigabit Ethernet or Infiniband. Each cluster can be heterogeneous (i.e., different hardware and software). Each node is managed by a virtualization solution such as Xen [8], KVM [15], OpenVZ [1] which supports VM live migration. VMs are seen as black-boxes which are organized in so-called virtual clusters (VC) where each VC represents a collection of one or multiple VMs hosting one or multiple applications. We assume no restriction about applications: both compute and server applications are supported. Multicast support is assumed to be available at network level. Consequently, complex network topologies supporting multiple clusters are supported given that multicast forwarding can be enabled on the routers. The current implementation of Snooze does not tolerate failures that partition the network. However appropriate leader election algorithms (e.g., [17]) can replace the implemented one to tolerate such failures. Finally hosts may fail, and failures are assumed to follow a fail-stop model.

2.2 Global System Overview

The global system overview of Snooze is shown in Figure 1. Note that for the ease of explanation this figure focuses on a single cluster while in a real scenario multiple clusters could be managed in the same manner. The architecture is partitioned into three layers: *physical*, *hierarchical*, and *client*. At physical layer, machines are organized in a cluster, in which each node is controlled by a so-called *Local Controller (LC)*.

A hierarchical layer allows to efficiently manage the cluster, and is composed of fault-tolerant components: *Group Managers (GMs)* and a *Group Leader (GL)*. Each GM manages a subset of LCs, and the GL keeps the summary information of the GMs.

Finally, a client layer provides the user interface. This interface is currently implemented by a predefined number of replicated *Entry Points (EPs)* and is queried by the clients in order to *discover the current GL*. Clients can interact with the EPs by utilizing the provided bindings. In order to provide a simple yet flexible interface, all system components are implemented as Java RESTful web services. In the following sections, the details of each component are discussed.

2.3 System Components

2.3.1 Local Controller (LC)

At the physical layer, the LC of a node is in charge of the following tasks: (1) joining the hierarchy during system boot and rejoining the hierarchy in case of GM failures, (2) performing total host capacity retrieval (total amount of

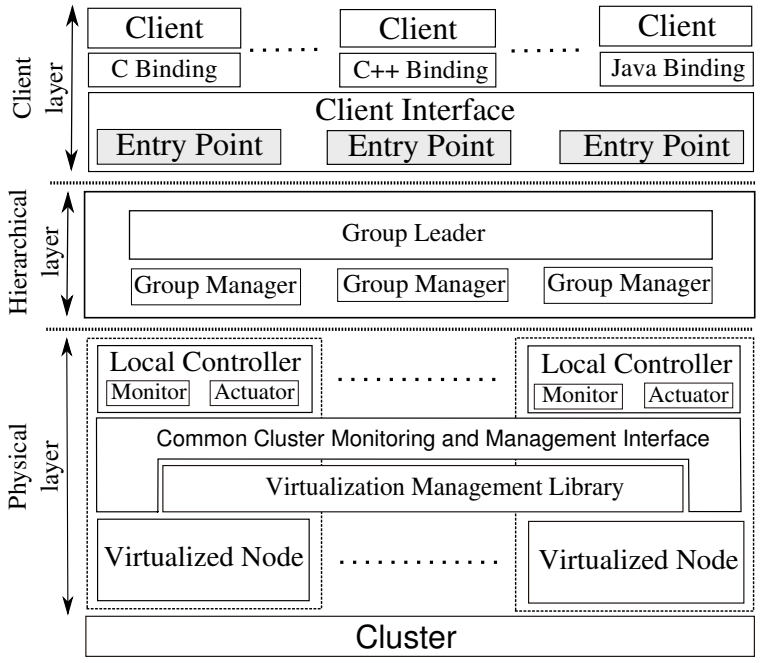


Figure 1: Global system overview

CPUs/cores, memory and network capacity), (3) performing VM monitoring (current CPU, memory, and network utilization), and (4) enforcing VM management commands (start, suspend, resume, save, restore, shutdown, destroy, resize and migrate).

Therefore, a LC has two components: *Monitor* and *Actuator*. The *Monitor* implements all the logic required to monitor the host and its VMs, including reporting this information to the assigned GM. Similarly, the *Actuator* enforces the VM management commands coming from the GM. Both components rely on the *Common Cluster Monitoring and Management Interface (CCMMI)*, which allows to support different virtualization solutions, like the *libvirt virtualization management library* used in the current implementation, as well as external monitoring frameworks such as Ganglia [4].

2.3.2 Group Managers (GMs)

Each node (i.e., LC) of the physical layer is managed by one of the GMs within the hierarchical layer (see Figure 2). This management involves six tasks: (1) receive, store, and answer queries for host and VM monitoring information, (2) estimate VM resource demands, (3) schedule VMs, (4) send VM management (e.g., start, stop) enforcement requests to the LCs, (5) transmit GM summary information to the GL and finally (6) announce its presence.

The host and VM monitoring information supporting the VM scheduling engine decisions is periodically sent to the GMs by the LCs and stored in an in-memory repository (other backends like Apache Cassandra [16] can be used to implement this repository).

Based on this monitoring information, VMs' resource demand estimates,

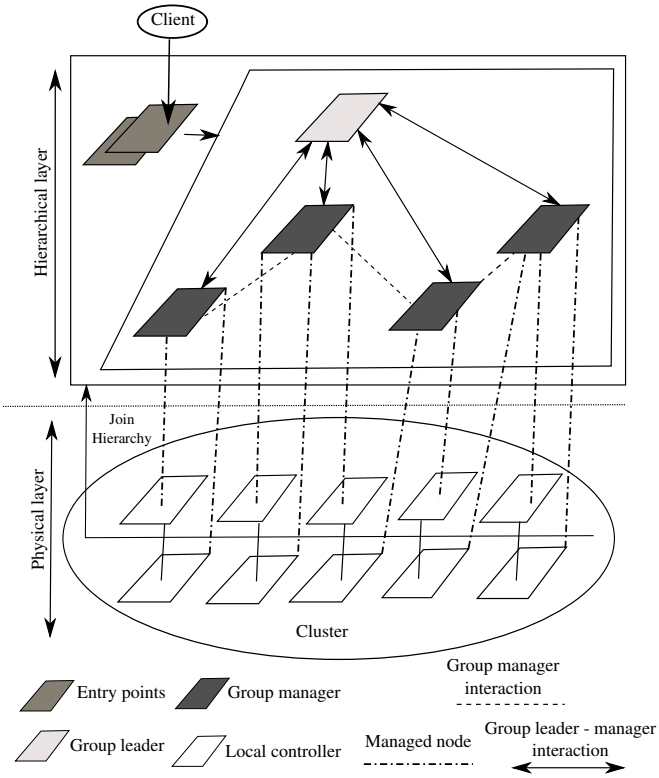


Figure 2: Hierarchical architecture overview

that are required by advanced optimization policies like VM consolidation, are performed by an integrated estimation engine using interfaces to support different CPU, memory, and network demand estimators. For instance, a Double-Exponential-Moving-Average (DEMA) (resp. Autoregressive-Moving-Average (ARMA))) estimator can be used to estimate CPU (resp. memory) demand.

VM scheduling is performed on each GM by a generic engine that currently distinguishes between three types of policies implemented by the administrator: (1) scheduling, (2) optimization, and (3) planning. While scheduling policies (e.g., round robin, load balance) are used for the incoming VMs, optimization and planning policies can be used to periodically optimize the VM placement. The specified optimization policy computes the optimized VM placement and the planning policy computes a migration plan which gives the order of migrations required to move the system from the current state to the optimized one. For example, given that the administrator enables continuous optimization and specifies the proper reconfiguration interval (e.g., daily at 1 AM), Snooze attempts to optimize the current VM placement periodically using the pluggable optimization policy (e.g., consolidation). To this end, the engine queries the estimation engine in order to get the current VM resource demand estimates, triggers the optimization and planning policies, and finally sends live migration requests to the actuators of the LCs. Note, that the evaluation of scheduling features it out of the scope of this paper.

GM summary information is periodically sent by each GM to the current GL

in order to support high-level scheduling decisions of the GL such as delegating VMs to be scheduled on the GMs, and to detect GM failures. This information is also used in case of GL failure to rebuild the GL system view. Currently, the GM summary includes the total amounts of used and free capacity available on all the managed LCs, as well as the VM networking related information (see the following sections for more details).

Finally, to support GM failure-detection, each GM is part of a heartbeat multicast group on which it periodically announces its presence.

2.3.3 Group Leader (GL)

The GL oversees the GMs and fulfills the following tasks: (1) stores incoming GM monitoring summary information, (2) dispatches incoming VC submission requests, (3) assigns joining local controllers to GMs, (4) manages VM network addresses, and (5) periodically announces its presence. Note, that unlike on GMs, only lightweight (i.e., dispatching) VM placement decisions are performed. Still, its scalability can be further improved with replication and an additional load balancing layer.

GM summary information is received and stored in the GL repository in order to guide VC dispatching as well as LCs to GM assignment. Therefore, the GL integrates a scheduling engine using two types of policies. VC dispatching policies take as input a VM description as well as a repository reference and output the GM to which the VM start request should be delegated. The GL then sends the VM start requests to the assigned GMs, waits for the replies and returns the output (i.e., status of VMs, assigned IP addresses, and GM descriptions) to the caller (see Section 2.3.5). LC assignment policies take as input a LC description as well as the GM repository reference and output the assigned GM. Passing this references allows to support advanced (e.g., based on GM load) assignment policies. A LC needs to know which GM it is assigned to in order to join the hierarchy (see Section 2.4 for the details of the join procedure).

VM network management is handled transparently for the user by Snooze. Each GL maintains a system administrator configurable subnet from which it is allowed to allocate IP addresses. When a VC is submitted to the GL, each of its VMs automatically gets an IP address from this subnet. The IP address is then encoded in the MAC address of the VM description before it gets dispatched to the GM. When the VM boots it executes a script which decodes the IP from the VM MAC address and performs the network configuration. A similar approach is applied in OpenNebula [18]. In order to let the GL recycle IP addresses, the summary information of a GM includes a list of the IP addresses of its managed VMs that have recently been terminated.

Finally, similar to GMs, the GL announces its presence on a predefined GL heartbeat multicast group.

2.3.4 Entry Points (EPs)

The EPs are used by the client software to discover the current GL. In order to keep track of the GL's address, all EPs subscribe to the GL heartbeat multicast group and listen for current GL announcements.

2.3.5 Command Line Interface (CLI)

A Java-based CLI is implemented on top of the RESTful interfaces exported by the EPs, GL and GMs. It supports the definition and management of VCs as well as visualizing and exporting the current hierarchy organization in the GraphML format.

When a user defines a VC and attempts to start it, the CLI first tries to transparently discover an active EP by walking through the EPs list specified in its configuration file and testing the EP status. Given that an active EP exists, a GL lookup is sent in order to receive the current GL information. Finally, the request to submit the VC is delegated to the GL which dispatches the VMs on the available GMs. The result is returned to the CLI and presented to the user. Currently, the following information is provided: Assigned VM IP addresses, GM descriptions (i.e., hosts and ports), status (e.g., RUNNING) and an error code which is displayed if problems occurred during the submission. Finally, the GM information on which the VMs were dispatched is stored in the local CLI repository, thus allowing the CLI to directly contact the GM whenever VC/VM management commands need to be performed. The following management commands are currently supported: starting, stopping, suspending, resuming, inter-GM migration, dynamic resizing (number of virtual cores and memory size) as well as retrieving the current live resource usage statistics of the VMs. Finally, it is important to mention that on GM failures the CLI repository information becomes obsolete. When the CLI detects that a GM is not reachable (e.g., during VM resource usage information retrieval) it first queries the EP in order to discover the current GL. Afterwards a GM discovery request including the VM identifier is sent to the GL. Upon reception of the request, the GL queries the currently active GMs in order to find the one assigned to the VM, and returns the result to the CLI. Thus the management command can be performed on the new GM.

2.4 Self-Organization and Self-Healing of the Hierarchy

GM join process and GL election When a new GM attempts to join the system, the leader election algorithm is triggered. Currently, our leader election algorithm is built on top of the Apache ZooKeeper [13] highly available and reliable coordination system, with its implementation following the recipe proposed by the authors of the service in [2].

ZooKeeper is integrated as follows. When Snooze is deployed, the ZooKeeper service is installed on the EPs in replication mode. The GM connects to the ZooKeeper service upon boot, creates an *ephemeral* node in its hierarchical namespace and attaches the GM description (i.e., networking information and an internal Snooze identifier) to it. This node is assigned a unique sequential identifier by the service in the namespace and is used by the GMs in order to discover the current GL and elect a new GL in case of failure. After the node creation each GM first tries to find another node in the namespace with a lower identifier (i.e., predecessor). If such a node already exists, the GM starts watching it and initiates the GL heartbeat multicast listener. Finally, upon reception of a GL heartbeat message, the GM sends a join request along with its description to the GL. Otherwise, if no node with a lower identifier could be detected, the current GM becomes the new GL and starts announcing its

presence by sending multicast messages on the GL heartbeat multicast port.

GL and GM failure recovery Each time a failure of a GL or GM occurs, an event is triggered on its successor GM as each node watches its predecessor. The successor GM becomes the new GL if its identifier is the lowest and stops all the GM related logic. Otherwise it simply starts watching the next predecessor GM. When a GM is promoted to be a GL, it gracefully terminates all its tasks such as the heartbeat and monitoring data sender, its open LC connections, and the repository manager. Afterwards the GL logic is started along with the GL heartbeat sender. Finally, as all the existing GMs are still listening for GL heartbeat messages they receive the new GL information and automatically trigger the GL rejoin procedure. Note that in case of a GL failure all its internal knowledge about the existing GMs as well as the distributed VM networking information (i.e., assigned IP addresses) is lost. In order to restore this knowledge, each time a GM rejoins the GL, the GM sends its description along with the VM networking information stored in its repository. Moreover, GM resource utilization summaries are periodically sent back to the GL, thus making it completely recover the system view.

Finally, as a GM has been promoted to become the new GL and thus has terminated all its GM related logic, LCs which were previously assigned to it fail to receive its heartbeat messages and trigger the system rejoin procedure.

LC join process and recovery The join process of a LC works as follows. Each time a LC attempts to join the hierarchy it starts listening for the GL multicast heartbeat messages. When it receives a heartbeat message, the join process is started by sending a GM assignment request with its current description (i.e., host address, port and total capacity) to the GL. The GL then triggers the GM assignment policy and dispatches the LC to an active GM. The contact information of the allocated GM is returned to the LC, which then initiates the actual GM join process by sending its description to the GM. Afterwards, it starts listening for GM heartbeat messages and periodically sends its own heartbeat, host and VM monitoring information to the GM. On an LC failure, the GM gracefully removes it from its database, and adds the IP addresses of the LC's VMs to the list of freed addresses. Note, that in case of LC failure, VMs executing on the LC are terminated. Therefore, *snapshot* features of hypervisors can be used by LCs in order to periodically save VM states (i.e., CPU, memory, disk) on stable storage on behalf of the GM. This will allow the GM to reschedule the failed VMs on its active LCs.

Finally, in case of a GM failure, the LC rejoins the hierarchy by triggering again the join procedure. Because VM information (i.e., identifier, assigned IP address, etc.) stored by the previous GM might get lost, each time a LC joins the newly assigned GM, it transfers its local state (i.e., information about currently running VMs), thus allowing the GM to update its repository. This update is needed by the clients to discover the new VM location (see Section 2.3.5) and by the GM to perform VM management operations as well as scheduling decisions.

3 Experimental Results

In order to evaluate the features of the Snooze framework we have deployed it on a 144 nodes cluster of the Grid'5000 experimental testbed in Nancy (France). Each node is equipped with one quad-core Intel Xeon X3440 2.54 GHz CPU, 16 GB of RAM, and a Gigabit Ethernet interconnect. The operating system on each server of the selected cluster is Debian with a 2.6.32-5-amd64 kernel. All tests were run in a homogeneous environment with qemu-kvm 0.14.1 and libvirt 0.9.6-2 installed on all machines. Each VM is using a QCOW2 disk image with the corresponding backing image hosted on a Network File System (NFS). Debian is installed on the backing image and uses a ramdisk in order to speed up the boot process. Finally, the NFS server is running on one of the EPs with its directory being exported to all LCs.

Our study is focused on evaluating the VM submission time in a centralized as well as distributed deployment, the impact of fault-tolerance on application performance, and finally the scalability of the framework.

3.1 Submission time: Centralized vs. Distributed

Submission time was evaluated by deploying a large amount of VMs and is defined as the time between initiating the submission request and receiving the reply on the client side. This involves assigning IP addresses, dispatching VMs to the GMs, scheduling VMs on the GMs and finally returning the response to the client.

Two deployment scenarios were created: *centralized* and *distributed*. In the former the EP, GL, GM as well as the Apache ZooKeeper service were running on the same machine while 136 nodes were hosting the LCs. This allowed us to reproduce the traditional frontend/backend-model as close as possible. In the latter, the system was configured in a distributed manner with two EPs, each of them hosting a replica of the Apache ZooKeeper service. In addition 6 GMs (including one GL) and 136 LCs were used. Finally, in both scenarios increasing numbers of VMs were submitted simultaneously to the system, the numbers varying from 0 to 500 in 50 VM steps. Each VM required one virtual core and 2 GB of RAM. All VM templates and disk images were pre-created on the NFS-server and submissions happened sequentially directly after the predecessor VMs were terminated. 500 VMs were a good tradeoff (i.e., $\sim$4 VMs per LC) in order not to risk application performance degradation due to possible resource overcommit.

The experimental results of this evaluation are plotted in Figure 3. As it can be observed, submission time increases approximately linearly with the amount of VMs in both the centralized and distributed deployment. However, more interesting is the fact that besides minor measurement errors, submission times in both scenarios are nearly equivalent thus indicating the good scalability of the system as no overhead of being distributed can be observed. Finally, submission of 500 VMs were finished in less than four minutes which indicates the good quality of our prototype implementation.

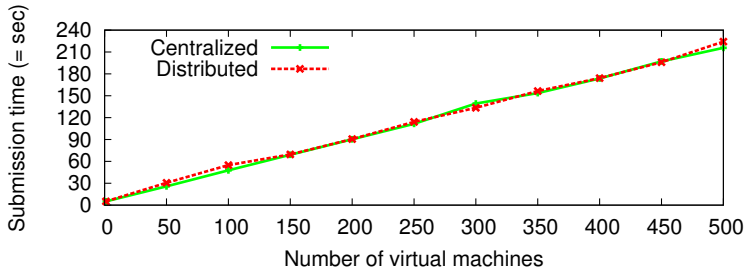


Figure 3: Submission time: Centralized vs. Distributed

3.2 Fault-Tolerance and Application Performance

To evaluate the impact of fault-tolerance on application performance the system was configured in a distributed manner (see Section 3.1). Two types of VMs with the following applications were created: (1) VMs hosting the MPI-implementation of the *NAS Parallel Benchmark (NPB) 3.3* [7], that represent high performance computing workloads and (2) VMs hosting the Linux, Apache, MySQL, PHP (LAMP) stack running the *Pressflow v6 content management system (CMS)* [6], that represent scalable servers workloads.

For MPI we have selected the FT benchmark from NPB, because of its heavy use of collective communication thus leading to high average network utilization (approximately 100 Mbit/s per VM). The benchmark was run with the Class A problem size across 100 VMs and the total execution time was measured. For web applications, the performance of *Pressflow v6* was analyzed while running on a dedicated VM by running the *Apache HTTP server benchmarking tool* [3] with concurrency set to 100 and number of requests to 1000. All measurements were repeated five times and the average values were taken.

To get an insight in the actual impact of fault-tolerance on application performance (i.e., execution time and throughput), system component failures were injected randomly in the middle of the benchmark execution. Three types of failures were injected: single GM failure, catastrophic GM failures (i.e., 1/2 of GMs fail) and finally a GL failure.

The results of this evaluation are shown in Figure 4 and 5. As it can be observed, apart from measurement errors neither in the MPI nor in the web-based benchmarks any performance degradation can be observed. This is not surprising as the heartbeat overhead is negligible (see Section 3.3). Moreover, due to the ZooKeeper service supporting GL election process, the amount of traffic in case of GL failure is low. Indeed, our leader election implementation only requires to watch (i.e., small heartbeat messages) the predecessor nodes (see Section 2.4). Similarly, the amount of data required for a GM to rejoin the new GL is approximately 100 bytes (i.e., GM host, port and networking information). Finally, in case of GM failures only small description information (i.e., LC host, port and VM descriptions) needs to be transferred by LCs to the current GL and the new GM thus not requiring substantial amounts of network capacity. Consequently, no overhead of fault-tolerance can be observed.

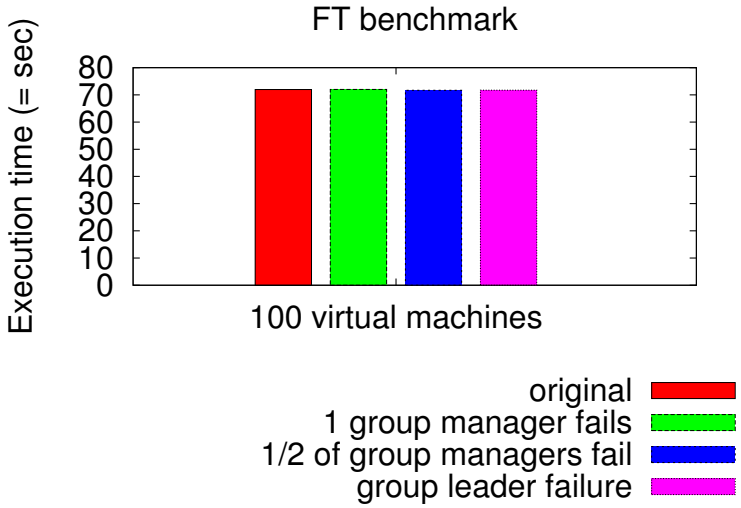


Figure 4: FT benchmark

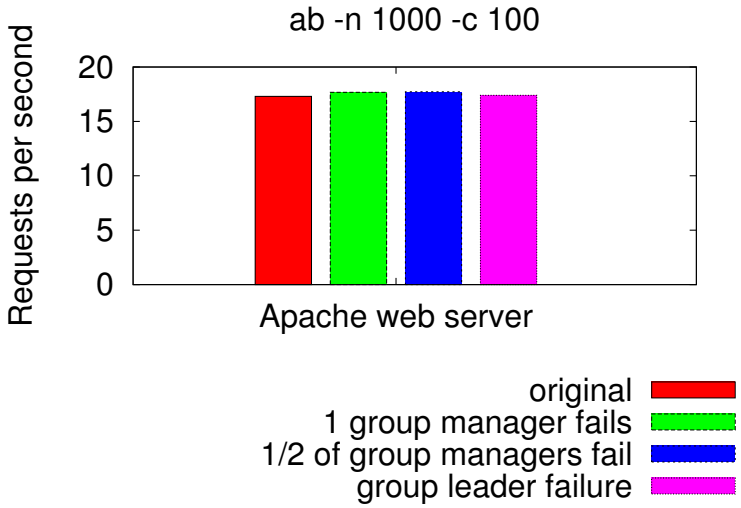


Figure 5: Pressflow CMS

3.3 Scalability of the Framework

The network load scalability of the framework was evaluated by measuring the network utilization at the GL, GM and LC. From these values we estimated the upper bounds for the network load scalability of the framework. Therefore, to isolate the heartbeat and monitoring traffic the framework was deployed with one EP, GL, GM and LC. Heartbeat intervals of the GL as well as of the GM were set to 3 seconds. Moreover, monitoring information summary was sent by the GM and LC periodically in 10 second intervals. Accounting the monitoring information is important as it is involved in the process of failure-detection (see

Section 2.4). During all tests, the LC assignment policy as well as the VC and VM scheduling policy were set to *round robin* thus resulting in a *balanced hierarchy* in terms of LC assignments as well as VM locations.

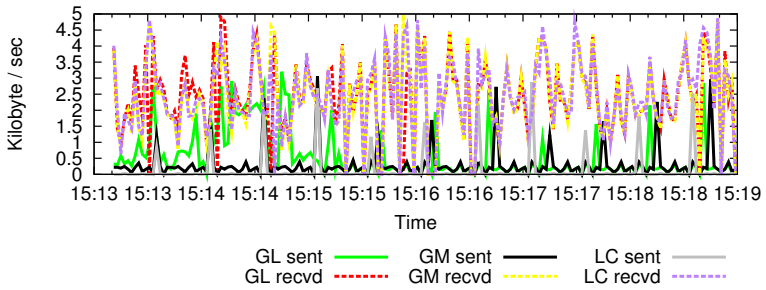


Figure 6: Network load scalability

Figure 6 depicts the correlated incoming and outgoing network traffic of the GL, GM and LC. As it can be observed, the heartbeat multicast messages of the GL only account to approximately 2.5 kB/s thus not putting any significant pressure on the network. On the other hand, GL incoming traffic is mainly dominated by the received GM monitoring summary information which amounts to approximately 4.5 kB/s and is sent using TCP sockets. Given that the summary information is of a fixed size and assumed a gigabit network interconnect, a GL could handle approximately up to *140,000* of GMs until its network capacity would become saturated. This theoretical upper bound is much higher than what any existing deployment would need.

When considering the network load scalability of the GMs, heartbeats are sent from the each GM to its LCs and vice versa. Analogously to the GL, GM heartbeat messages are multicast based while LC monitoring information is periodically sent using TCP sockets. For scalability and system design reasons, only one TCP connection exists per LC to its assigned GM over which all host, VM and heartbeat monitoring information is sequentially transmitted. Thus when no VMs are active, still a fixed amount of data (i.e., heartbeat) is periodically sent by each LC. This amount of data can be observed in Figure 6. Particularly, as the LC monitoring information is of the same structure as the one from a GM, approximately 4.5 kB/s are arriving at the GM. Similarly, the heartbeat information sent by the GM and GL is equivalent in terms of size (i.e., ∼2.5 kB/s). Putting all these facts together and making the same assumptions about the network bandwidth, similar amount (i.e., ∼140,000) of LCs could be handled by each GM. Finally, when considering the network load scalability upper bound of the GL as well as of the GMs, theoretically Snooze would be able to manage up to *20 billions* of compute nodes (i.e., LCs).

We now discuss the CPU and memory load scalability of the GL as well as of the GMs. Figure 7 first presents the GL CPU and memory statistics obtained during the distributed VM submission time evaluation (see paragraph 3.1).

We notice that there is a short spike in CPU load and memory usage at the beginning of the experiment. Indeed the GL service needs to be started first. Afterwards, the actual VC submission is started. After the boot period the system settles at a fixed memory amount of approximately 327 MB (includ-

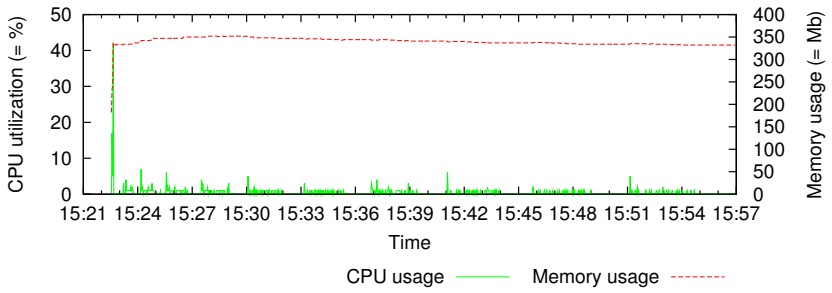


Figure 7: GL CPU and memory during VM submission

ing OS services) which remains constant with the number of VMs submitted. Similarly, small CPU load spikes can be clearly observed during periods of VM submissions which are as well independent of the VMs amount and never exceed 10% of CPU utilization. Both results emphasize the good scalability of the GL.

In order to get more insights about the GL as well as GM scalability with increasing amount of resources we have evaluated our system with varying numbers of GMs and LCs. In the first experiment, the amount of LCs was fixed and the number of GMs was dynamically doubled every minute until 128 GMs were researched. Same study was conducted with a single GM by increasing the amount of LCs up to 128 and measuring the resulting overheads.

Figure 8 depicts the results from the first evaluation.

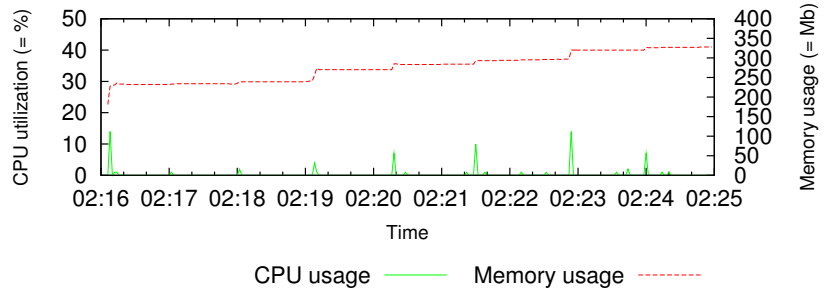


Figure 8: GL CPU and memory during VM submission

While the system scales well with respect to CPU utilization (i.e., small spikes during GM joins), because GM summaries are stored in-memory, the memory usage increases linearly with the number of GMs. Note, that this bottleneck can be resolved by providing a different implementation for the repository interface (e.g., based on Apache Cassandra [16]).

Figure 9 presents the results from the second experiment. Apart from a similar repository bottleneck, the system shows good CPU load scalability with increasing number of LCs.

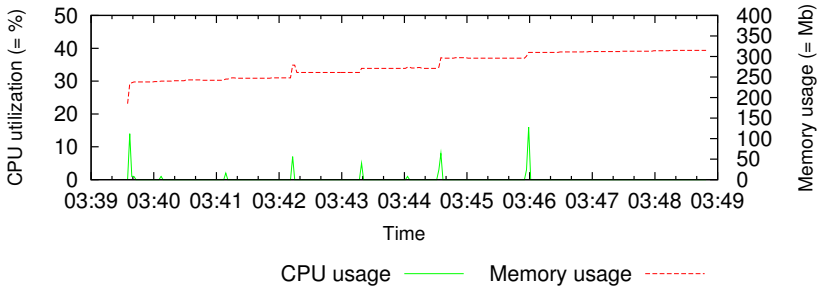


Figure 9: GM CPU and memory with increasing amount of LCs

4 Related Work

Several VM management frameworks such as OpenNebula [18], Nimbus [14], Eucalyptus [20], and OpenStack [5] have been developed during the last years. Thereby, as previously discussed in Section 1, the architectures of the first three frameworks follow the traditional frontend/backend model and thus suffer from limited scalability and SPOF.

Eucalyptus is the closest open-source system in terms of architecture. Similar to the architecture of Snooze, a higher instance (i.e., cloud controller) oversees the underlying *cluster controllers*. The cluster controllers on the other hand are in charge of managing the compute resources each running a *node controller*. However, unlike in Snooze where a group manager is designed to manage a *subset* of each cluster, each *cluster controller* in Eucalyptus is designed to manage the *entire cluster* thus making it suffer from SPOF. Moreover, Eucalyptus does not include any self-healing features and strictly distinguishes between cloud and cluster controllers (i.e., static hierarchy) while Snooze follows a more self-organizing approach in which each group manager (GM) is promoted to a group leader (GL) dynamically during the leader election procedure and upon GL failure detection. In addition, cluster controllers of Eucalyptus do not support VM resource (i.e., CPU utilization, memory and network) monitoring. Thus they are limited to simple static VM scheduling policies (e.g., greedy, round robin). On the contrary, in Snooze each GM is designed to periodically receive VM resource usage information from the local controllers (LCs).

Recently in [23] a more distributed peer-to-peer (P2P) based VM scheduling approach is introduced. However, this work is still in very early stages as it is limited to a single load balancing policy and no evaluation regarding its scalability and fault-tolerance aspects is performed. Finally, only limited simulation-based results are presented. Another VM management framework based on a P2P network of nodes is presented in [22]. The nodes are organized in a ring and scheduling is performed iteratively upon underload and overload events triggered by the nodes. However, neither the overhead of maintaining the ring structure nor the scalability and fault-tolerance aspects are discussed. Similarly to the previous work only preliminary simulation-based results targeting the scheduling time are presented. In contrast to these two works, nodes in Snooze are dynamically organized in a self-healing hierarchical architecture. This allows it to scale with the increasing number of nodes (i.e., GL does not require global knowledge) as well as to provide the required fault-tolerance prop-

erties without the need to rely on P2P technology. In fact, our experimental results show that a lightweight approach taken in our work is sufficient in order to provide scalability and fault-tolerance properties for data center sizes which go beyond existing deployments. Moreover, Snooze is a working implementation which provides most of the features (i.e., client interface, support for different scheduling policies, virtual network management, etc) required for it to be called a *management framework*. Last but not least, it is evaluated in a real environment and shown to be scalable and fault-tolerant. Still, it is clear that the scalability of our system is bounded by its GL which acts as a fault-tolerant lightweight central point. Despite the fact that GL can be replicated and a load balancing layer can be added to improve its scalability, when developed and evaluated, fully decentralized P2P-based systems could turn out to be even more scalable alternatives.

Besides the existing VM management frameworks, more generic frameworks targeting scalability and fault-tolerance issues in distributed systems have been proposed in the past. Particularly, the hierarchical layer of Snooze is inspired from the idea introduced in the Hasthi [21] framework which takes a hierarchical self-stabilizing approach for managing large-scale distributed systems. Through simulations the authors show that their system is able to scale up to 100,000 resources. Contrary to Hasthi whose design is presented to be system agnostic and utilizes a distributed hash table (DHT) based peer-to-peer (P2P) network, Snooze follows a simpler design and does not require the use of P2P technology. Moreover, it targets virtualized platforms and thus its design and implementation is driven by the platform specific objectives and issues. Finally, Snooze has a working implementation which is evaluated in a real environment. Similarly to Hasthi our system is shown to achieve excellent scalability.

Another system based on a hierarchical architecture is called DIET [10]. However, DIET implements the GridRPC programming model while Snooze targets VM management. Consequently, both frameworks are orthogonal. For example, virtual resources of Snooze can be used by the DIET framework in order to perform computation tasks.

5 Conclusions and Future Work

In this paper, we have presented a *novel scalable and fault-tolerant* VM management framework called *Snooze*. Unlike the previous open-source virtual infrastructure managers, Snooze employs a self-organizing hierarchy in which the VM management tasks are distributed across multiple self-healing *group managers (GMs)* with each GM having only a partial view of the system. Thereby, each GM is only in charge of managing a *subset* of the compute nodes (i.e., local controllers (LCs)). Moreover, a group leader (GL) dispatches VM submission requests and handles the VM networking. This allows our framework to stay highly scalable as the GL does not require any global system knowledge. In addition, the GL facilitates the system management by playing the role of a fault-tolerant coordinator. Such a coordination helps keeping the design and implementation simple. Finally, to the best of our knowledge this is the first work to analyze the load scalability and fault-tolerance of an IaaS-cloud computing framework in a real environment.

Our extensive evaluation conducted on the Grid'5000 experimental testbed

has shown that: (1) submission time is not impacted by performing distributed VM management, (2) systems fault-tolerance properties do not impact application performance and finally (3) the system scales well with increasing number of resources thus making it suitable for managing large-scale virtualized data centers. Finally, thanks to the integrated VM monitoring and resource demand estimation support as well as the generic scheduling engine, the system is well suitable as a research testbed to design, implement and evaluate advanced VM placement policies in a real environment.

In the future, we plan to conduct a performance comparison of Snooze with existing open-source cloud management frameworks. Recent evaluating efforts (e.g., [25], [24]) mainly focus on the functional aspects. Moreover, the system will be made even more autonomic by removing the distinction between GMs and LCs. Consequently, the decisions when a node should play the role of GM or LC in the hierarchy will be taken by the framework instead of the system administrator upon configuration. In addition, we will improve the scalability of the database and GL. Particularly, current in-memory repository implementation will be replaced by a distributed NoSQL database (i.e., Apache Cassandra). Scalability of the GL will be improved by adding replication and additional load balancing layer. Finally, the NFS-based VM image storage will be replaced by a distributed file system (e.g., BlobSeer [19]).

Another important aspect of our work is *power management*. Snooze already has experimental support for power management and VM consolidation which will be evaluated along with the integration of our recently proposed nature-inspired VM placement algorithm [11] in the near future. Ultimately, Snooze will become an open-source project.

6 Acknowledgment

We wish to express our gratitude to Stefania Costache, Piyush Harsh, and Pierre Riteau for their detailed feedback. This research is funded by the french *Agence Nationale de la Recherche (ANR)* project EcoGrappe under the contract number ANR-08-SEGI-000. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] OpenVZ. 2.1

[2] ZooKeeper recipes and solutions. 2.4

[3] ab - Apache HTTP server benchmarking tool, 2011. 3.2

[4] Ganglia monitoring system, 2011. 2.3.1

[5] OpenStack: Open source cloud computing software, 2011. 1, 4

[6] Pressflow: Distribution of drupal with integrated performance, scalability, availability, and testing enhancements, 2011. 3.2

[7] David Bailey, John Barton, and Horst Simon. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63, 1991. 3.2

[8] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003. 2.1

[9] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *GRID '05: Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106, 2005. 1

[10] Eddy Caron and Frédéric Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006. 4

[11] Eugen Feller, Louis Rilling, and Christine Morin. Energy-aware ant colony based workload placement in clouds. In *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing (GRID-2011)*, Lyon, France, September 2011. 5

[12] Eugen Feller, Louis Rilling, Christine Morin, Renaud Lottiaux, and Daniel Leprince. Snooze: A scalable, fault-tolerant and distributed consolidation manager for large-scale clusters. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 125–132, 2010. 1

[13] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *In USENIX Annual Technical Conference*, 2010. 2.4

[14] Kate Keahey, Tim Freeman, Jerome Lauret, and Doug Olson. Virtual workspaces for scientific applications. *Journal of Physics: Conference Series*, 78(1):012038, 2007. 1, 4

[15] Avi Kivity. kvm: the Linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007. 2.1

[16] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a P2P network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009. 2.3.2, 3.3

[17] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, May 1998. 2.1

[18] Dejan Milojicic, Ignacio M. Llorente, and Ruben S. Montero. OpenNebula: A cloud management tool. *IEEE Internet Computing*, 15:11–14, March 2011. 1, 2.3.3, 4

[19] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. BlobSeer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.*, 71:169–184, February 2011. 5

[20] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, 2009. 1, 4

[21] Srinath Perera and Dennis Gannon. Enforcing user-defined management logic in large scale systems. In *Proceedings of the 2009 Congress on Services - I*, pages 243–250, Washington, DC, USA, 2009. IEEE Computer Society. 4

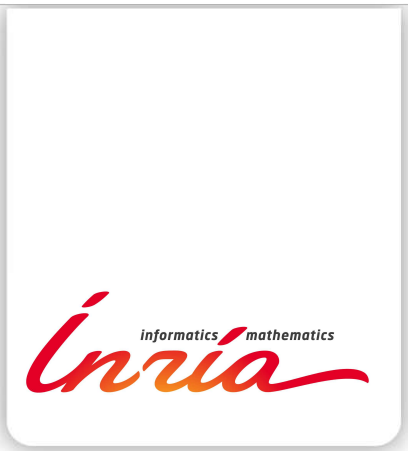
[22] Flavien Quesnel and Adrien Lèbre. Cooperative dynamic scheduling of virtual machines in distributed systems. In *Proceedings of the 6th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '11) Euro-Par 2011*, Bordeaux, France, August 2011. 4

[23] Jonathan Rouzaud Cornabas. A distributed and collaborative dynamic load balancer for virtual machine. In *Proceedings of the 5th Workshop on Virtualization in High-Performance Cloud Computing (VHPC '10) Euro-Par 2010*, Ischia, Naples Italy, 2010. 4

[24] P. Sempolinski and D. Thain. A comparison and critique of Eucalyptus, OpenNebula and Nimbus. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 417 –426, 2010. 5

[25] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13:14–22, September 2009. 5

Contents



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399