VMPlaceS A Generic Tool to Investigate and Compare VM Placement Algorithms

Adrien Lebre[1], Jonathan Pastor[1], Mario Südholt1

ASCOLA Research Group, Mines Nantes / Inria / LINA, Nantes, France
`firstname.lastname@inria.fr`

Most current infrastructures for cloud computing leverage static and greedy policies for the placement of virtual machines. Such policies impede the optimal allocation of resources and the satisfaction of operational guarantees like service-level agreements. In recent years, more dynamic and often more efficient policies based, *e.g.*, on consolidation and load balancing techniques, have been developed. Due to the underlying complexity of cloud infrastructures, these policies are evaluated either using limited scale testbeds/*in-vivo* experiments or ad-hoc simulator techniques. These validation methodologies are unsatisfactory for two important reasons: they (i) do not model precisely enough real productions platforms (size, workload representativeness, failure, etc.) and (ii) do not enable the fair comparison of different approaches.

In this article, we propose VMPlaceS, a dedicated simulation framework to perform in-depth investigations of VM placement algorithms and compare them in a fair way. Built on top of the SimGrid simulation platform, our framework provides programming support to easy the implementation of placement algorithms and runtime support dedicated to load injection and execution trace analysis. VMPlaceS supports a large set of parameters enabling researchers to design simulations representative of real-world scenarios. We also report on a validation of our framework by evaluating three classes of placement algorithms: centralized, hierarchical and fully-distributed ones. We show that VMPlaceS enables researchers (i) to study and compare such strategies, (ii) to detect possible limitations at large scale and (iii) easily investigate different design choices.

1 Introduction

Even if more flexible and often more efficient approaches to the Virtual Machine Placement Problem (VMPP) have been developed , most of the popular Cloud Computing (CC) system management [1,?,?], *a.k.a.* Infrastructure-as-a-Service toolkits [16], continue to rely on elementary Virtual Machine (VM) placement policies that prevent them from maximizing the usage of CC resources while guaranteeing VM resource requirements as defined by Service Level Agreements (SLAs). Typically, a batch scheduling approach is used: VMs are allocated according to user requests for resource reservations and tied to the nodes where they were deployed until their destruction. Besides the fact that users often overestimate their resource requirements, such static policies are definitely not optimal for CC providers, since the effective resource requirements of each operated VM may significantly vary during its lifetime.

An important impediment to the adoption of more advanced strategies such as consolidation, load balancing and other SLA-ensuring algorithms that have been deeply investigated by the research community [11,?,?,?,?,?] is related to the experimental processes that have been used to validate them: most VMPP proposals have been evaluated either by leveraging ad-hoc simulators or small testbeds. These evaluation environments are not accurate and not representative enough to (i) ensure their correctness on real platforms and (ii) perform fair comparisons between them.

Implementing each proposal and evaluating it on representative testbeds in terms of scalability, reliability and varying workload changes would definitely be the most rigorous way to observe and propose appropriate solutions for CC production infrastructures. However, *in-vivo* (*i.e.*, real-world) experiments, if they can be executed at all, are always expensive and tedious to perform (for recent reference see [5]). They may even be counterproductive if the observed behaviors are clearly different from the expected ones.

In this article, we propose VMPlaceS, a dedicated simulation framework to perform in-depth investigations of VM placement algorithms and compare them in a fair way. To cope with real conditions such as the increasing scale of modern data centers and the dynamicity of the workloads that are specific to the CC paradigm, notably its elasticity capacity, VMPlaceS allows users to study large-scale scenarios that take into account server crashes and that involve tens of thousands of VMs, each executing a specific workload that evolves during the simulation lifetime.

Built on top of the SimGrid toolkit [10], VMPlaceS provides three additional simulation facilities: a more abstract representation of hosts and virtual machines, and two frameworks, one for the management of load injection abstract and another one for the extraction and analysis of execution traces. We believe that such a tool will be beneficial to a large number of researchers in the field of CC as it enables them to quickly validate the trends of a new proposal and compare it with existing ones. This way, our approach allows *in vivo* experiments to be restricted to VMPP mechanisms that have the potential to handle CC production infrastructures.

We chose to base VMPlaceS on SimGrid since (i) the latter's relevance in terms of performance and validity has already been demonstrated [3] and (ii) because it has been recently extended to integrate virtual machine abstractions and a live migration model [14].

To illustrate the relevance of VMPlaceS, we have implemented the essential mechanisms of three well-known VMPP approaches: Entropy [13], Snooze [11], and DVMS [18]. Besides being well-known from the literature, we chose these three systems as they are built on three different software architecture approaches: Entropy relies on a centralized model, Snooze on a hierarchical one and DVMS on a fully distributed one.

Once the accuracy of VMPlaceS validated, we have also investigated the characteristics of these three strategies by studying their scalability, reliability and reactivity (*i.e.*, the time to solve resource violations, *a.k.a.* SLA violation) criterions through several simulations. This study reveals:

– The importance of the duration of the reconfiguration phase (*i.e.*, the step where VMs are relocated throughout the infrastructure) in comparison to the computation one (*i.e.*, the step where the scheduler try to solve the VMPP, see Section **??** for a complete definition of both steps).
– The pros and cons of partitioning Snooze in small or large groups;
– The relevance of a reactive strategy in comparison to a periodic one.
– The interest of leveraging our toolkit to identify limitations of proposals and study variants and possible improvements.

The rest of the article is organized as follow. Section **??** highlights the importance of the scalability, reliability and reactivity criterions for the VM Placement Problem. Section 2 gives an overview of the SimGrid framework on which our proposal is built. 3 introduces VMPlaceS and discusses its general functioning. The three algorithms implemented as use-cases are presented in Section 4 and evaluated in Section 5. Section 6 and Section 7 present, respectively, related work as well as a conclusion and future work.

2 Simgrid, a generic toolkit

We now briefly introduce the toolkit on which VMPlaceS is based. SimGrid is a toolkit for the simulation of potentially complex algorithms executed on large-scale distributed systems. Developed for more than a decade, it has been used in a large number of studies described in more than 100 publications. Its main characteristics are the following:

– Extensibility: after Grids, HPC and P2P systems, SimGrid has been recently extended with abstractions for virtualization technologies (*i.e.*, Virtual Machines including a live migration model [14]) to allow users to investigate Cloud Computing challenges [15].
– Scalability: it is possible to simulate large-scale scenarios; as an example, users can simulate applications composed of 2 million processors and an infrastructure composed of 10,000 servers [10].
– Flexibility: it enables simulations to be run on arbitrary network topologies under dynamically changing computations and available network resources.
– Versatile APIs: users can leverage SimGrid through easy-to-use APIs for C and Java.

To perform simulations, users should develop a *program* and define a *platform* file and a *deployment* file. The *program* leverages, in most cases, the SimGrid MSG API that allows end-users to create and execute SimGrid abstractions such as processes, tasks, VMs and network communications. The *platform* file provides the physical description of each resource composing the environment and on which aforementioned computations and network interactions will be performed in the SimGrid world. The *deployment* file is used to launch the different SimGrid processes defined in the *program* on the different nodes. Finally, the execution of the program is orchestrated by the SimGrid engine that internally relies on an constraint solver to correctly assign the amount of CPU/network resources to each SimGrid abstraction during the entire simulation.

SimGrid provides many other features such as model checking, the simulation of DAGs (Direct Acyclic Graphs) or MPI-based applications. In the following, we only give a brief description of the virtualization abstractions that have been recently implemented and on which our framework relies on (for further information regarding SimGrid see [10]).

The VM support has been designed so that all operations that can be performed on a host can also be performed inside a VM. From the point of view of a SimGrid Host, a SimGrid VM is an ordinary task while from the point of view of a task running inside a SimGrid VM, a VM is considered as an ordinary host. SimGrid users can thus easily switch between a virtualized and non-virtualized infrastructure. Moreover, thanks to MSG API extensions, users can control VMs in the same manner as in the real world (*e.g.*, create/destroy VMs; start/shutdown, suspend/resume and migrate them). For migration operations, a VM live migration model implementing the precopy migration algorithm of Qemu/KVM has been integrated into SimGrid. This model is the only one that successfully simulates the live migration behavior by taking into account the competition arising in the presence of resource sharing as well as the memory refreshing rate of the VM, thus determining correctly the live migration time as well as the resulting network traffic [14]. These two capabilities are mandatory to build our VM placement simulator toolkit.

3 VM Placement Simulator

The purpose of VMPlaceS is to deliver a generic tool to evaluate new VM placement algorithms and offer the possibility to compare them. Concretely, it supports the management of VM creations, workload fluctuations as well as node apparitions/removals. Researchers can thus focus on the implementation of new placement algorithms and evaluate how they behave in the presence of changes that occur during the simulation. VMPlaceS has been implemented in Java by leveraging the messaging API (MSG) of SimGrid. Although the Java layer has an impact of the efficiency of SimGrid, we believe its use is acceptable because Java offers important benefits to researchers for the implementation of advanced scheduling strategies, notably concerning the ease of implementation of new strategies. As examples, we reimplemented the Snooze proposal in Java and the DVMS proposal using Scala/Java.

In the following we give an overview of the framework and describe its general functioning.

3.1 Overview

From a high-level view, VMPlaceS performs a simulation in three phases: (i) initialization (ii) injection and (iii) trace analysis (see Figure 1). The initialization phase corresponds to the creation of the environment, the VMs and the generation of the queue of events that may represent, *e.g.*, load changes. The simulation is performed by at least two SimGrid processes, one executing the *injector*, which constitutes the generic part of the framework, and a second one executing the to-be-simulated *scheduling algorithm*. During the simulation the scheduling strategy is evaluated by injecting scheduling-relevant events.

AL⇒MS This sentence above is not accurate enough I? previously it was : The *injector* constitutes the generic part of the framework. It injects scheduling-relevant events during the execution of simulations. We should reformulate

Currently, the supported events are VM CPU load change and node apparitions/removals that we use to simulate node crashes.

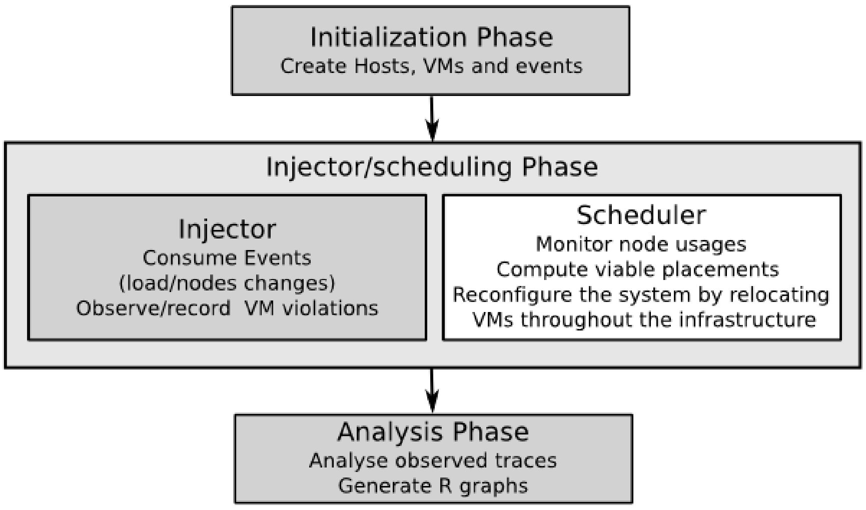

Initialization Phase
Create Hosts, VMs and events
Injector/scheduling Phase
Injector
Consume Events
(load/nodes changes)
Observe/record VM violations
Scheduler
Monitor node usages
Compute viable placements
Reconfigure the system by relocating VMs throughout the infrastructure
Analysis Phase
Analyse observed traces
Generate R graphs

Fig. 1. VMPlaceS's Workflow
Gray parts correspond to the generic code while the white one must be provided by end-users. The current version is released with three different schedulers (centralized/hierarchical and distributed).

Users develop their scheduling algorithm by leveraging the SimGrid messaging API and a more abstract interface that is provided vy VMPlaceS and consists of the classes `XHost`, `XVM` and `SimulatorManager` classes. The two former classes respectively extend SimGrid's `Host` and `VM` abstractions while the latter controls the interactions between the different components of the VM placement simulator. Throughout these three classes, users can inspect, at any time, the current state of the infrastructure (*i.e.*, the load of a host/VM, the number of VMs hosted on the whole infrastructure or on a particular host, check whether a host is overloaded, etc.) We have used VMPlaceS in order to analyze three scheduling mechanisms, cf. Sec. 4, that represent three different software architecture models: centralized, hierarchical and fully-distributed models for VM placement.

The last phase consists in the analysis of the collected traces in order to gather the results of the simulation, notably by means of the generation of figures representing, *e.g.*, resource usage statistics.

3.2 Initialization Phase

In the beginning, VMPlaceS creates n VMs and assigns them in a round-robin manner to the first p hosts defined in the platform file. The default platform file corresponds to a cluster of $h + s$ hosts, where h corresponds to the number of hosting nodes and s to the number of services nodes. The values n, h and s constitute input parameters of the simulations (specified in a Java property file). These hosts are organized in form of topologies, a cluster topology being the most common ones. It is possible, however, to define more complex platforms to simulate, for instance, scenarios involving federated data centers.

Each VM is created based on one of the predefined VM classes. A VM class corresponds to a template specifying the VM attributes and its memory footprint. Concretely, it is defined in terms of five parameters: the number of cores `nb_cpus`, the size of the memory `ramsize`, the network bandwidth `net_bw`, the maximum bandwidth available migrate it `mig_speed` and the maximum memory update speed `mem_speed` when the VM is consuming 100% of its CPU resources. As pointed out in Section 2, the memory update speed is a critical parameter that governs the migration time as well as the amount of transferred data. By giving the possibility to define VM classes, VMPlaceS allows researchers to simulate different kinds of workload (*i.e.*, memory-intensive vs non-intensive workloads), and thus analyze more realistic Cloud Computing problems. Available classes are defined in a specific text file that can be modified according to the user's needs. Finally, all VMs start with a CPU consumption of 0 that will evolve during the simulation in terms of the injected load as explained below.

Once the creation and the assignment of VMs completed, VMPlaceS spawns at least two SG processes, the *injector* and the launcher of the selected scheduler. The first action of the *injector* consists in creating the different event queues and merge them into a global one that will be consumed during the second phase of the simulation. For now, we generate two kinds of event: *CPU load* and *node crash* events. The *CPU load* event queue is generated in order to change the load of each VM every t seconds on average. t is a random variable that follows an exponential distribution with rate parameter λ_t while the CPU load of a VM evolves according to a Gaussian distribution defined by a particular mean (μ) as well as a particular standard deviation (σ). t, μ and σ are provided as input parameters of a simulation. As the CPU load can fluctuate between 0 and 100%, VMPlaceS prevents the assignment of nonsensical values when the Gaussian distribution returns a number smaller than 0 or greater than 100. Although this has no impact on the execution of the simulation, we emphasize that this can reduce/increase the effective mean of the VM load, especially when σ is high. Hence, it is important for users to specify appropriate values. Furthermore, each random process used in VMPlaceS is initialized with a seed that is defined in a configuration file. This way, we can ensure that different simulations are reproducible and may be used to establish fair comparisons.

The *node crash* event queue is generated in order to turn off a node every f seconds on average for a duration of d seconds. Similarly to the t value above, f follows an exponential distribution with rate λ_f. f and d are also provided as input parameters of a simulation.

Adding new events can easily be done by simply defining new event Java classes implementing the `InjectorEvent` interface and by adding the code in charge of generating the associated queue. Such a new queue will be merged into the global one and its events will then be consumed similarly to other ones during the *injector phase*.

3.3 Injector Phase

Once the VMs and the global event queue are ready, the evaluation of the scheduling mechanism can start. First, the injector process iteratively consumes the different events that represent, for now, load changes of a VM or turning a node off or on. Changing the load of a VM corresponds to the creation and the assignment of a new SimGrid task in the VM. This new task has a direct impact on the time that will be needed to migrate the VM as it increases or decreases the current CPU load and thusits memory update speed. When a node is turning off, the VMs that were running on that node are temporarily discarded, *i.e.*, they are hidden and cannot be accessed until the node comes back to life. This way, the scheduler cannot handle them. We leave for future work other approaches that can better match realistic scenarios such as turning off the VMs and reprovisioning them on other nodes.

As defined by the scheduling algorithm, VMs will be suspended/resumed or relocated on the available hosts to meet scheduling objectives and SLA guarantees. Note that users must implement the algorithm in charge of solving the VMPP but also the code in charge of applying reconfiguration plans by invoking the appropriate methods available from the `SimulatorManager` class. This step is essential as the reconfiguration cost is a key element of dynamic placement systems.

Last but not least, it is noteworthy that VMPlaceS really invokes the execution of each scheduling strategy in order to get the effective reconfiguration plan. That is, the computation time that is observed is not simulated but corresponds to the effective one, only the workload inside the VMs and the migration operations are simulated in SimGrid. It is hence mandatory to propagate the reconfiguration time into the SimGrid engine.

3.4 Trace Analysis

The last step of VMPlaceS consists in analyzing the information that has been collected during the simulation. This analysis is done in two steps. First, VMPlaceS records several metrics related to the platform utilization throughout the simulation by leveraging an extended version of SimGrid's TRACE module[1]. This way, visualization tools that have been developed by the SimGrid community, such as PajeNG [2], may be used. Furthermore, our extension enables the creation of a trace file in the JSON file format, which is used to generate several figures using the R statistical environment [7] about the resource usage during the simulation.

By default, VMPlaceS records the load of the different VMs and hosts, the appearance and the duration of each violation of VM requirements in the system, the number of migrations, the number of times the scheduler mechanism has been invoked and the number of times it succeeds or fails to resolve non-viable configurations. Although these pieces of information are key elements to understand and compare the behavior of the different algorithms, we emphasize that the TRACE API enables the creation of as many variables as necessary, thus allowing researchers to instrument their own algorithm with specific variables that record other pieces of information.

4 Dynamic VMPP Algorithms

To illustrate the interest of VMPlaceS, we implemented three dynamic VM placement mechanisms respectively based on the Entropy [13], Snooze [11], and DVMS [18] proposals. For the three implementations, we chose to use the latest VMPP solver that has been developed as part of the Entropy framework [12].

Giving up consolidation optimality in favor of scalability, this algorithm provides a "repair mode" that enables the correction of VM requirement violations. The algorithm considers that a host is overloaded when the VMs try to consume more than 100% of the CPU capacity of the host. In such a case, the algorithm looks for an optimal viable configuration until it reaches a predefined timeout. The optimal solution is a new placement that satisfies the requirements of all VMs while minimizing the cost of the reconfiguration. Once the timeout has been triggered, the algorithm returns the best solution among the ones it finds and applies the associated reconfiguration plan by invoking live migrations in the simulation world.

Although using the Entropy VMPP solver implies a modification from the original Snooze proposal, we highlight that our goal is to illustrate the capabilities of VMPlaceS and thus we believe that such a modification is acceptable as it does not change the global behavior of Snooze. Moreover by conducting such a comparison, we also investigate the pros and cons of the three architecture models on which these proposals rely on (*i.e.*, centralized, hierarchical and distributed).

Before discussing the simulation results, we describe in this section an overview of the three implemented systems. We highlight that the extended abstractions for hosts (`XHost`) and VMs (`XVM`) as well as the available functions of the SimGrid MSG API enabled us to develop them in a direct and natural manner.

[1] http://simgrid.gforge.inria.fr/simgrid/3.12/doc/tracing.html

4.1 Entropy-based Centralized Approach

The centralized VM placement mechanism consists in one single SimGrid process deployed on a service node. This process implements a simple loop that iteratively checks the viability of the current configuration by invoking every p seconds the aforementioned VMPP solver. p is defined as an input parameter of the simulation.

As the Entropy proposal does not provide a specific mechanism for the collect of resource usage information but simply uses an external tool (namely ganglia), we had two different ways to implement the monitoring to process: either by implementing additional asynchronous transmissions as a real implementation of the necessary state updates would proceed or, in a much more lightweight manner, through direct accesses by the aforementioned process to the states of the hosts and their respective VMs. While the latter does not mimic a real implementation closely, it can be harnessed to yield a valid simulation: overheads induced by communication in the "real" implementation, for instance, can be easily added as part of the lightweight simulation. We have implemented this lightweight variant for the monitoring

Regarding fault tolerance, similarly to the Entropy proposal, our implementation does not provide any failover mechanism.

Finally, as mentioned in Section 3.4, we monitor, for each iteration, whether the VMPP solver succeeds or fails. In case of success, VMPlaceS records the number of migration that has been performed, the time it took to apply the reconfiguration and whether the application of the reconfiguration plan led to new violations.

4.2 Snooze-based Hierarchical Approach

We now present Snooze [11] as a second case study of how to implement and simulate advanced algorithms. We present its architecture summarizing its main characteristics from its original presentation [11] and additional information stemming from personal communications of the Snooze developers and its implementation [4,20].

Architecture Snooze harnesses a hierarchical architecture in order to support load balancing and fault tolerance, cf. Fig. 2. At the top of the hierarchy, a *group leader (GL)* centralizes information about the whole cluster using summary data about *group managers (GMs)* that constitute the intermediate layer of the hierarchy. GMs manage a number of *local controllers (LCs)* that, in turn, manage the VMs assigned to nodes. The GL and the GMs are deployed on service nodes while the LCs are executed on hosting node. During execution, higher-level components periodically send heartbeats to lower-level ones; monitoring information, *e.g.*, about the system load, is also sent periodically in the opposite direction. In order to propagate information down the hierarchy, Snooze relies on hardware support for multicast communication. Finally, a number of replicated

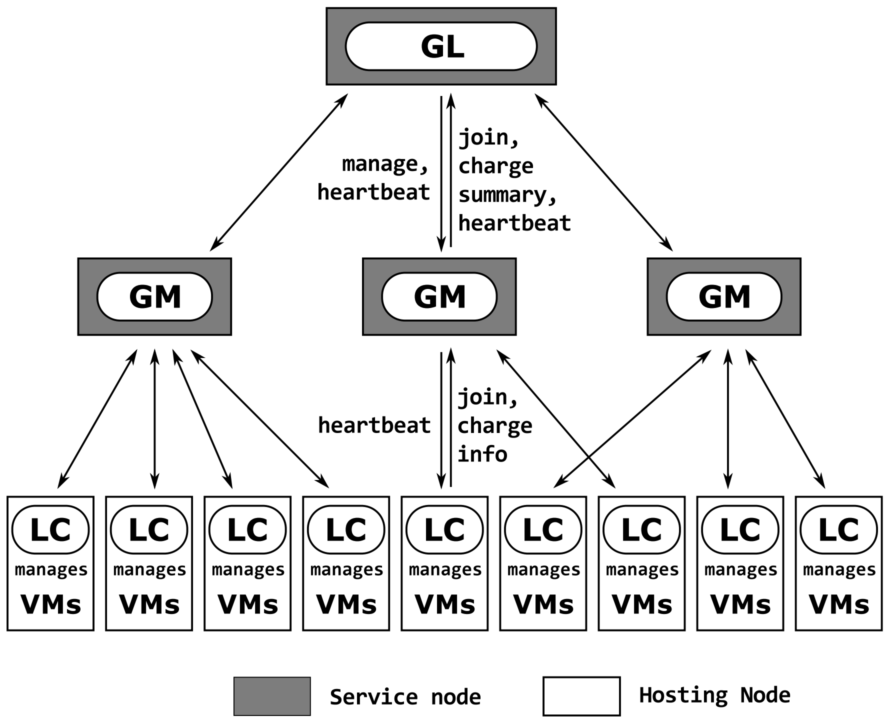


Fig. 2. Overview of Snooze's architecture

entry points allows clients to contact the GL, *e.g.*, in order to submit new VMs for integration into the system.

Simulation using VMPlaceS. The `XHOST`, `XVM` and `SimulatorManager` classes have been harnessed to implement the core architectural abstractions (*i.e.*, VM monitoring and manipulations), the remaining concepts and algorithms of Snooze have been implemented using Simgrid's primitives and standard Java mechanisms. Communication between Snooze actors is implemented based on Simgrid's primitives for, mainly asynchronous, event handling. Hardwaresupported multicast communication that is used, *e.g.*, to relay heartbeats, is implemented as a dedicated actor that manages a state representing GL and GM heartbeat groups and relaying heartbeat events. Finally, our Snooze simulation uses, as its original counterpart, a multi-threaded implementation (*i.e.*, based on multiple SG processes) in order to optimize reactivity even for large groups of LCs (or GMs) that have to be managed by one GM (or GL).

Algorithms Apart from the handling of faults (described below), two types of algorithms are of major importance for the administration of the Snooze architecture: the algorithms that enable components to dynamically enter the system and the algorithms that propagate info between the components.

A GL is created, if it does not exist, by promotion of a GM that is selected according to some leader election algorithm. When a GM joins a cluster, it starts listening on a predefined channel for the heartbeat of the GL and registers once it has received the heartbeat. New LCs first also wait for the GL heartbeat, contact the GL then in order to obtain a GM assignment, and finally register at the GM assigned to them.

Two kinds of (load) information are passed within the system: the periodic heartbeat message sent by the GL and the GMs; second, periodic load information sent from LCs to their respective GMs and summary load info sent by the GMs to the GL.

Fault tolerance GLs, GMs and LCs may fail during the system execution. System components identify that a node on the corresponding higher-level node has failed (the GL in case of a GM, a GM in the case of an LC) in an asynchronous fashion through the lack of heartbeat messages.

In the case of a GL failure, one of the GMs becomes the new GL, stops its GM activities and prevents the LCs it manages so that they can start rejoining the system. If a GM fails, the GL and the LCs it has managed will become aware of it based on the lack of heartbeats, update its data structures and, for the LCs, rejoin the system. If an LC fails, its GM will finally learn of it due to the missing heartbeat and charge information of the LC. The GM will then remove the LC from its data structures.

4.3 DVMS-based Distributed Approach

As the third use-case, we have implemented the DVMS (Distributed Virtual Machine Scheduler) proposal for the cooperative and fully distributed placement of VMs [18].

Architecture A DVMS agent is deployed on each node in order to manage the VMs on the node and collaborate with (the agents of) neighboring nodes. Agents are defined on top of an overlay communication network, which defines the node-neighbor relation and can be structured (using, *e.g.*, Chord [19]) or unstructured. For this study, we have implemented a simple but effective unstructured overlay that enables the agents to collaborate without side effects: when necessary, *e.g.*, in case of node failures, the overlay provides a link to a neighbor of a node on the latter's request.

When a server is overloaded (*i.e.*, VMs hosted on the server require more resources than available), an *Iterative Scheduling Procedure (ISP)* is started: a partition initially containing only the overloaded node is created; the partition then grows by including free nodes until the resource requirements can be satisfied by a VM reconfiguration. This way, resource problems that appear at many different nodes can be handled in parallel using different partitions.

Each partition includes two special nodes, the initiator and the leader. The initiator of a partition is its initial node (i.e. the overloaded node). The leader of a partition is the node that was the last to be added to the partition: it manages the scheduling computations necessary to resolve the overload resource conflict. If a valid reconfiguration plan cannot be computed, a new node will be inserted in the partition, who becomes the new leader of the partition.

Iterative Scheduling Procedure When a node N$_i$ detects that it cannot provide enough resources for its hosted VMs, it generates a partition and reserves itself to solve the problem (see Figure 3(a)) and thus initiates a partition. Then, its closest neighbor, as defined by the network overlay, is considered.

If this neighbor, N$_{i+1}$, is already part of another partition, its neighbor is considered. Otherwise, N$_{i+1}$ joins the partition (see Figure 3(b)). If the partition is not valid anymore (*e.g.*, because the workload of the partition's VM has decreased), N$_{i+1}$ cancels the reservations, destroys the partition and thus frees its nodes for another problem solving procedure. On the contrary, if the procedure is still valid, N$_{i+1}$ notifies members of the partition that it has become the new leader. The other nodes then send it information about their capacities and current load. The leader, in turn, starts a scheduling computation looking for a reconfiguration within the current partition. If no solution is found, the same algorithm is applied to the next node N$_{i+2}$. In the extreme case a partition may grow until all resources in a cluster contribute to the resolution of its resource scheduling problem. This approach harnesses as few nodes as possible, thus accelerating the scheduling computations to the maximum possible.

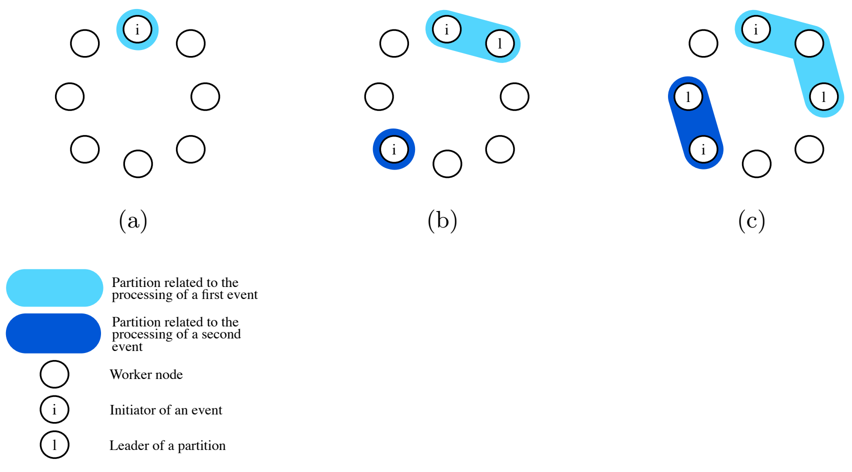

i
(a)
i
l
i
(b)
i
l
l
i
(c)
Partition related to the processing of a first event
Partition related to the processing of a second event
Worker node
Initiator of an event
Leader of a partition
(d) Legend

Fig. 3. Processing two events simultaneously

Fault-tolerance The main advantage of using overlay networks is that they have built-in fault tolerance mechanisms. DVMS therefore works on top of an overlay network such as Chord: when a node needs to rebalance its VMs workload, it uses the overlay network to find collaborators. For this study we implemented a simple overlay network as a flat list of agents: a typical request for collaborators includes the list of agents that are already collaborating with the requesting agent. A link to a new collaborator is then provided to the requesting agent. Communication is performed by message exchanges containing immutable data: our implementation harnesses the principles of the actor model in order to ease the handling of concurrency and distributed issues.

Harnessing the fault tolerance mechanisms of the underlying overlay network is, however, not sufficient. If the leader of a partition crashes, a new leader must take over in order for the resource problem to be solved and the nodes of a partition to be finally freed. To avoid these issues, DVMS now relies on timeout mechanisms. Each node of a partition periodically checks whether the state of its partition changed recently (*e.g.*,, if a new node joined the partition) and can thus identify if the partition's leader is not active anymore. In this case, each node leaves the partition and can be integrated in other partitions.

5 Experiments

In this section, we, first, analyze the accuracy of VMPlaceS by comparing the results of an Entropy execution through simulations and *in-vivo* experiments. This first experiment enables us to confirm the expected behavior of VMPlaceS Second, we present and discuss our analysis of the three algorithms previously described. We show that the performances of the hierarchical approach could reached the distributed ones through minor changes.

5.1 A First Use-Case: Comparison of Entropy, Snooze and DVMS

In this section, we discuss the results of the simulations we performed on the Entropy, Snooze and DVMS strategies. First, we present a general study analyzing the violation times as well as the duration of the computation and reconfiguration phases. Second, we examine some variants and possible improvements of Snooze and DVMS that made possible to easily study thanks to VMPlaceS.

Experimental Conditions All simulations have been performed on the Lyon clusters of the Grid'5000 testbed. Each execution was running on a dedicated server, thus avoiding interferences between simulations and ensuring reproducibility between the different invocations.

VMPlaceS has been configured in order to simulate an homogeneous infrastructure of PMs composed of 8 cores, 32 GB of RAM and 1 Gpbs Ethernet NIC. To enable fair comparison between the three strategies, the scheduling resolver only considered 7 cores (*i.e.*, one was devoted to run the Snooze LC or the

DVMS processes). Dedicating one core for the host OS and other administrative processes is something which is rather usual and thus we believe acceptable in our experimental methodology. Regarding the Virtual Machines, Ten VMs have been initially launched on each simulated PM. Each VM relied one of the VM classes previously described in Section **??** and the parameters for changing the load were the same (λ = Nb VMs/300, $\mu = 60$ and $\sigma = 20$). The stationary state was reached after 20 min of the simulated time with a global load of 85% as depicted in Fig. 4(a). To accelerate the simulations, we have chosen to limit the simulated time to 1800 seconds. It is noteworthy that the consolidation ratio, *i.e.*, the number of VMs per node, has been defined to generate a sufficient number of violations. We discovered that under a global load of 75%, our infrastructure almost did not face VM violations with our selected Gaussian distribution. Such a result is rather satisfactory as it can explained why most production DCs target such an overall utilization rate.[2]

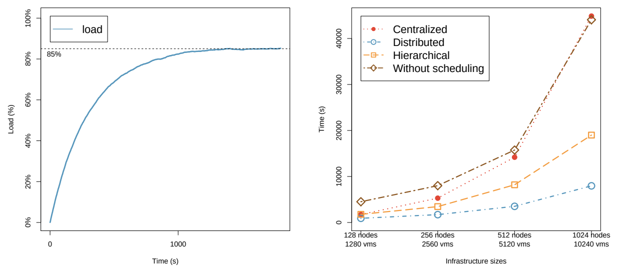
We conducted simulations in order to study infrastructures composed of 128, 256, 512 and 1024 PMs, hosting respectively 1280, 2560, 5120 and 10240 VMs. Additional simulated PMs have been provided to execute the Entropy and Snooze service nodes on distinct nodes. For Snooze, one GM has been created every 32 LCs (*i.e.*, PMs). Entropy and Snooze are invoked every 30 seconds. Finally, it is noteworthy that no service node had to be provisioned for DVMS as a DVMS process had been executed directly on top of the hosting nodes.

In order to cope with real DC conditions, we defined the parameters for node crashes to simulate a fault on average every 6 months for a duration of 300 seconds. These values correspond to the Mean Time To Failure (MTTF) and the Mean Time To Repair (MTTR) of a Google DC server [6, pp. 107-108]. We underline that at the scale we performed our simulations such a crash ratio was not sufficient to impact the behavior of the scheduling policies. Dedicated simulations were mandatory to study the influence of node crashes. However, due to the space limitations, we do not present them in the article and only gives major trends. Regarding Entropy, although the lost of the service node can be critic, the failure probability is so small that the single point of failure issue can be easily solved by a fail-over approach. Regarding Snooze, the heartbeat strategy enables the reconstruction of the hierarchy in a relative short time and thus crashes on service nodes do not significantly impact the resolution of violations (in our case less than 10 seconds is mandatory to reorganize the Snooze topology with a 6 seconds heartbeat mechanism). Finally regarding DVMS, the crash of one node does not have any impact of the resolution has the composition of the microcosm is reevaluated immediately.

All configuration files used to perform the discussed simulations can be downloaded from the VMPlaceS repository.

General Analysis Fig. 4(b) presents the cumulated violation time for each placement policy while Tables 1, 2 and 3 give more details by presenting the

[2] http://www.cloudscaling.com/blog/cloud-computing/amazons-ec2-generating-220m-annually/



(a) Infrastructure load (b) Cumulated Violation Time

Fig. 4. Simulation Results - 10 VMs per node (VM load: $\mu = 60$ and $\sigma = 20$)

mean and the standard deviations of the duration of, respectively, the violations, the computation and reconfiguration phases. As anticipated, the centralized approach did not scale and became almost counterproductive for the largest scenario in comparison to a system that did not use any dynamic scheduling strategy. The more nodes Entropy has to monitor, the less efficient it is on both the computation and reconfiguration phases. Regarding the computation, the VMPP is a NP-Hard problem and thus it is not surprising that it takes more time to resolve larger problems. Regarding the reconfiguration, as Entropy has to solve much more violations simultaneously, the reconfiguration plan is more complex for large scenarios, including several migrations coming from and going to the same nodes. Such reconfiguration plans are non optimal as they increase the bottleneck effects at the network level of each involved PM. Such a simulated result is valuable as it confirms that reconfiguration plans should avoid as much as possible such manipulations.

Infrastructure Size	**Algorithm**		
	Centralized	Hierarchical	Distributed
128 nodes	21.26 ± 13.55	21.07 ± 12.32	9.55 ± 2.57
256 nodes	40.09 ± 24.15	21.45 ± 12.10	9.58 ± 2.51
512 nodes	55.63 ± 42.26	24.54 ± 16.95	9.57 ± 2.67
1024 nodes	81.57 ± 86.59	29.01 ± 38.14	9.61 ± 2.54

Table 1. Duration of violations ($Med \pm \sigma$)

Regarding Snooze, although the performances are better than the Entropy ones, we may erroneously conclude that the hierarchical approach is not competitive with respect to the distribued strategy at the first sight. However, diving into details, we can see that both the computation and reconfiguration phases are almost constants (around 3 seconds and 10 seconds) and not so far from the

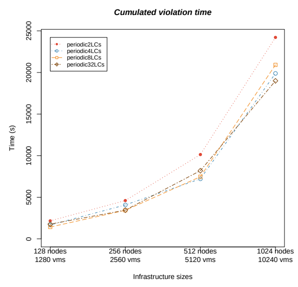
Infrastructure Size	**Algorithm**		
	Centralized	Hierarchical	Distributed
128 nodes	3.76 ± 7.43	2.52 ± 4.63	0.29 ± 0.03
256 nodes	7.97 ± 15.03	2.65 ± 4.69	0.25 ± 0.02
512 nodes	15.71 ± 29.14	2.83 ± 4.98	0.21 ± 0.01
1024 nodes	26.41 ± 50.35	2.69 ± 4.92	0.14 ± 0.01

Table 2. Duration of computations ($Med \pm \sigma$)

Infrastructure Size	**Algorithm**		
	Centralized	Hierarchical	Distributed
128 nodes	10.34 ± 1.70	10.02 ± 0.14	10.01 ± 0.11
256 nodes	10.26 ± 1.45	10.11 ± 0.83	10.01 ± 0.08
512 nodes	11.11 ± 3.23	10.28 ± 1.50	10.08 ± 0.82
1024 nodes	18.90 ± 7.57	10.30 ± 1.60	10.04 ± 0.63

Table 3. Duration of reconfigurations ($Med \pm \sigma$).

DVMS values, especially for the reconfiguration phase, which is predominant. These results can be easily explained: the centralized policy adresses the VMPP by considering all nodes at each invovation, while the hierarchical and the distributed algorithms divide the VMPP into sub problems, considering smaller numbers of nodes (32 PMs in Snooze and 4 in average with DVMS). To clarify the influence of the group size on the Snooze performances, *i.e.*, the ratio of LCs attached to one GM, we performed additional simulations aiming at investigating whether a smaller group size can lead to similar performances of DVMS. We higlight that the use of VMPlaceS eased such a study as it has consisted to simply relaunch the previous simulation with a distinct assignment.



(a) Total Violation Times

Infra. Size	**No. of failed reconfigurations**			
	2 LCs	4 LCs	8 LCs	32 LCs
128	19	0	0	0
256	29	0	0	0
512	83	1	0	0
1024	173	7	0	0

(b) No. of Failed Reconfigurations

Infra. Size	**Algorithm**			
	2 LCs	4 LCs	8 LCs	32 LCs
128	0.16 ± 1.23	0.34 ± 1.81	0.58 ± 2.40	2.53 ± 4.62
256	0.18 ± 1.31	0.42 ± 1.99	0.66 ± 2.50	2.65 ± 4.69
512	0.15 ± 1.20	0.33 ± 1.78	0.67 ± 2.54	2.83 ± 4.98
1024	0.19 ± 1.37	0.42 ± 2.02	0.89 ± 2.90	2.69 ± 4.91

(c) Means ± Std deviations of computation durations.

Fig. 5. Hierarchical placement: influence of varying group sizes

Varying group sizes Figure 5 presents the simulated values obtained for scenarios with 2, 4, 8 and 32 LCs per GM for four infrastructure sizes. The overall performance (*i.e.*, cumulated violation time), shown in Fig. 5(a), shows that 2 LCs per GM result in significantly higher violation times. All other group sizes yield violation times that are relatively close, which indicates that a small group size does not help much in resolving violations faster.

The relatively bad performance of the smallest group size can be explained in terms of the number of failures of the reconfiguration process, that is, overloading situations that are discovered but cannot be resolved because the GM managing the overloaded VM(s) did not dispose of enough resources, see Table 5(b). Groups of 2 LCs per GM are clearly insufficient at our load level (60% mean, 20% stddev). Failed reconfigurations are, however, already very rare in the case of 4 LCs per GM and do not occur at all for 8 and 32 LCs per GM. This is understandable because the load is statistically evenly distributed among the LCs and tthe load profile we evaluated only rarely results in many LCs of a GM to be overloaded. Violations can therefore be resolved even in the case of a smaller number (4) LCs available for load distribution.

Conversely, we can see that the duration of the overall reconfiguration phases decreases strongly along with the group size. It reaches a value close to the computation times of DVMS for a group size of 4-LCs per GM, see Fig. 5(c). We thus cannot minimize computation times and violation times by reducing the number of LCs because larger group sizes are necessary to resolve overload situations if the VM load gets higher. Once again, this information is valuable as it will help researchers to design new algorithms favoring the automatic discovery of the optimal subset of nodes capable to solve violations under for given load profiles.

In contrast, DVMS resolves this trade-off by construction because of its automatic and dynamic choice of the partition size necessary to handle an overload situation.

Another important metric when performing dynamic relocations is related to the migration overhead on the network but also on the workload running inside each VM. Hence it is important to understand how VM Placement algorithms deal with such a concern.

AL⇒AL finalized that part with the following paragraph

Figure 5.1 describes the cumulated migrations number for each strategy: the distributed algorithm spend more time on migrating VMs, followed by the hierarchical scheduler. It is noticeable that the centralized scheduler performes spend less time in migration of VMs. Thanks to figure 5.1, we can see that the average duration of a migration increases dramatically with the centralized policy at large scale (10240 VMs), while it remains stable with the distributed and hierarchical schedulers. This is due to the fact that the unique node invocking Entropy does not find the best reconfiguration in an adequate time, and must confine itself to a non optimal reconfiguration. These non optimal reconfigurations usually contain several migrations to the same nodes, thus leading to network congestion and increasing the average migration duration.

AL⇒MS,AL TODO find the best place to add the following paragraph

To conclude, although the simulations discussed in this article are limited to 10K VMs, we succeeded to conduct DVMS simulations including up to

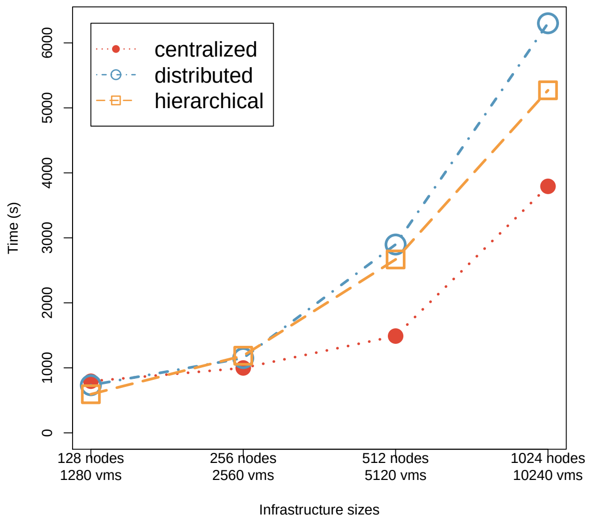


Fig. 6. Cumulated migration time.

8K PMs/80K VMs in a bit less than two days. We did not present such results to the paper because it was not possible to run a sufficient number of Snooze simulations at such scale. The Snooze protocol being more complex than the DVMS one (heartbeats, consensus, . . .), the time to perform a similar experiment is much more important (around 7 days). The time-consuming portions of the code are related to SimGrid internals such as `sleep` and `send/recv` calls. Hence, we have contacted the SimGrid core developers in order to investigate how we can reduce the required time to perform such advanced simulations.

6 Related Work

AL⇒AL Add few lines related to Google scheduling simulator - https://code.google.com/p/cluster-scheduler-simulator/

Several simulator toolkits have been proposed since the last years in order to adress CC concerns [9,**?**,**?**,**?**,**?**]. They can be classified into two categories: The first one corresponds to ad-hoc simulators that have been developped to address a particular concern. For instance, CReST [9] is a discrete event simulation toolkit built to evaluate Cloud provisioning algorithms. If ad-hoc simulators enable to provide some trends regarding the bevahiours of the system, they do consider the implication of the different layers, which can lead to non representative results at the end. Moreover, such ad-hoc solutions are developped for one shot and

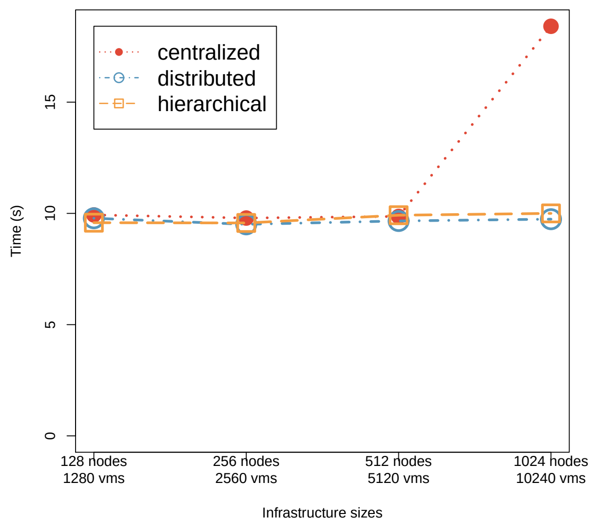


Fig. 7. Average duration of a migration.

thus, they are not available for the scientific community. The second category [17,**?**,**?**] corresponds to more generic cloud simulator toolkits (*i.e.*, they have been designed to adress a majority of CC challenges). However, they have focused mainly on the API and not on the model of the different mechanisms of CC systems.

For instance, CloudSim [8], which has been widely used to validate algorithms and applications in different scientific publications, is based on a relatively top-down viewpoint of cloud environments. That is, there is no papers that properly validate the different models it relies on: a migration time is calculated by dividing a VM memory size by a network bandwidth. In addition to having inaccuracy weaknesses at the low level, available cloud simulator toolkits over simplified the model for the virtualization technologies, leading also to non representation results at the end. As highlighted several times throughout this document, we chose to build VMPlaceS on top of SimGrid in order to benefit fromt its accuracy of its models related to virtualization abstractions [14].

7 Conclusion

In this paper we have presented VMPlaceS, a framework providing programming support for the definition of VM placement algorithms, execution support for their simulation at large scales, as well as new means for their trace-based analysis. VMPlaceS enables, in particular, the investigation of placement algorithms in the context of numerous and diverse real-world scenarios. We have validated its accuracy of returned results by comparing simulated and *in-vivo* executions of the Entropy strategy on top of 32 PMS and 192 VMs. We have illustrated the relevancce of VMPlaceS by evaluating and comparing algorithms representative for three different classes of virtualization environments: centralized, hierarchical and fully distributed placement algorithms. We have also shown how VMPlaceS facilitates the implementation and evaluation of variants of placement algorithms. The corresponding experiments have provided the first systematic results comparing these algorithms in environments including up to one thousand of nodes and ten thousands of VMs.

A beta-version of VMPlaceS is available on a public git repository[3]. We are in touch with the SimGrid core developers in order to improve ou code with the ultimate objective of adressing infrastructures up to 100K PMs and 1 Millions VMs. As future work, it would be valuable to add additional dimensions in order to simulate other workload variations stemming from network and HDD I/O changes. Finally, we plan to provide a dedicated API to be able to provision and remove VMs during the execution of a simulation.

Acknowledgments. 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. CloudStack, Open Source Cloud Computing. http://cloudstack.apache.org
2. PajeNG - Trace Visualization Tool. https://github.com/schnorr/pajeng
3. Simgrid publications. http://simgrid.gforge.inria.fr/Publications.html
4. Snooze web site, http://snooze.inria.fr, Last access: 21 Oct. 2014
5. Barker, A., Varghese, B., Ward, J.S., Sommerville, I.: Academic Cloud Computing Research: Five Pitfalls and Five Opportunities. In: Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud) (Jun 2014), https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/barker
6. Barroso, L.A., Hlzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2009)
7. Bloomfield, V.A.: Using R for Numerical Analysis in Science and Engineering. Chapman & Hall/CRC (2014), http://www.crcpress.com/product/isbn/9781439884485
8. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software: Practice and Experience 41(1), 23–50 (2011), http://dx.doi.org/10.1002/spe.995

[3] http://beyondtheclouds.github.io/VMPlaceS/

9. Cartlidge, J., Cliff, D.: Comparison of cloud middleware protocols and subscription network topologies using crest, the cloud research simulation toolkit - the three truths of cloud computing are: Hardware fails, software has bugs, and people make mistakes. In: CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science, Aachen, Germany, 8-10 May, 2013. pp. 58–68 (2013)
10. Casanova, H., Giersch, A., Legrand, A., Quinson, M., Suter, F.: Versatile, scalable, and accurate simulation of distributed applications and platforms. Journal of Parallel and Distributed Computing 74(10), 2899–2917 (Jun 2014), http://hal.inria.fr/hal-01017319
11. Feller, E., Rilling, L., Morin, C.: Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds. In: CCGRID '12: 12th Int. Symp. on Cluster, Cloud and Grid Comp. pp. 482–489 (May 2012)
12. Hermenier, F., Demassey, S., Lorca, X.: Bin Repacking Scheduling in Virtualized Datacenters. In: CP '11: Proceedings of the 17th international conference on Principles and practice of constraint programming. pp. 27–41. Springer (2011)
13. Hermenier, F., Lorca, X., Menaud, J.M., Muller, G., Lawall, J.: Entropy: A consolidation manager for clusters. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. pp. 41–50. VEE '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1508293.1508300
14. Hirofuchi, T., Lebre, A., Pouilloux, L.: Adding a live migration model into simgrid: One more step toward the simulation of infrastructure-as-a-service concerns. In: Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01. pp. 96–103. CLOUDCOM '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/CloudCom.2013.20
15. Lucas-Simarro, J.L., Moreno-Vozmediano, R., Desprez, F., Rouzaud-Cornabas, J.: Image transfer and storage cost aware brokering strategies for multiple clouds. In: IEEE CLOUD 2014. 7th IEEE International Conference on Cloud Computing. IEEE Computer Society, Anchorage, USA (June 27-July 2 2014)
16. Moreno-Vozmediano, R., Montero, R., Llorente, I.: IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. Computer 45(12), 65–72 (2012)
17. Nunez, A., Vazquez-Poletti, J., Caminero, A., Castañé, G., Carretero, J., Llorente, I.: icancloud: A flexible and scalable cloud infrastructure simulator. Journal of Grid Computing 10(1), 185–209 (2012), http://dx.doi.org/10.1007/s10723-012-9208-5
18. Quesnel, F., Lebre, A., Südholt, M.: Cooperative and Reactive Scheduling in Large-Scale Virtualized Platforms with DVMS. Concurrency and Computation: Practice and Experience 25(12), 1643–1655 (Aug 2013)
19. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. pp. 149–160. SIGCOMM '01, ACM, New York, NY, USA (2001)
20. Team, S.D.: Snooze characteristics and implementation (Jul 2014), personal communication