VMPlaceS – A Generic Tool to Investigate and Compare VM Placement Algorithms

Adrien Lebre, Jonathan Pastor, Mario Südholt
Ascola Research Group
Inria, Lina, Mines Nantes
Nantes, France
firstname.lastname@inria.fr

ABSTRACT

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.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

ACM proceedings, LaTeX, text tagging

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, 2, 6], *a.k.a.* Infrastructure-as-a-Service toolkits [20], 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 [12, 14, 16, 22, 25, 26] 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 [7]). 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 of which executes a specific workload that evolves during the simulation lifetime.

AL⇒MS each of which executing?

Built on top of the SimGrid toolkit [11], 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 [4] and (ii) because it has been recently extended to integrate virtual machine abstractions and a live migration model [15].

To illustrate the relevance of VMPlaceS, we have implemented the essential mechanisms of three well-known VMPP approaches: Entropy [14], Snooze [12], and DVMS [22]. 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.

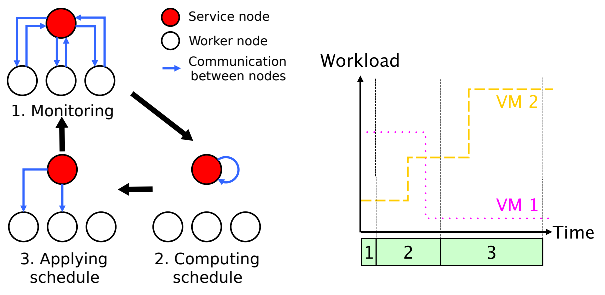
Once the accuracy of VMPlaceS validated, we have also investigated the characteristics of these three strategies by studying their scalability, reliability and reactivity (*i.e.*, the time to solve resource violations, *a.k.a.* SLA violation) criterions through several simulations. This study reveal:

- 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 2 for a complete definition of both steps).
- The quasi-inexistant impact of failures on the reactivity for all systems, especially if we are considering a 6 month failure rate;
- The pros and cons of partitionning Snooze in small or large groups;
- The pros and cons of a reactive vs a periodic scheduling strategies.
- 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 2 highlights the importance of the scalability, reliability and reactivity criterions for the VM Placement Problem. Section 3 gives an overview of the SimGrid framework on which our proposal is built. 4 introduces VMPlaceS and discusses its general functioning. The three algorithms implemented as use-cases are presented in Section 5 and evaluated in Section 6. Section 7 and Section 8 present, respectively, related work as well as a conclusion and future work.

2. THE VM PLACEMENT PROBLEM

A VMPP can be summarized in three-steps (see Figure 1(a)): monitoring the resources usages, computing a new schedule each time is it needed and applying the resulting reconfiguration plan (*i.e.*, performing VM migration and suspend/resume operations to realize to the new placement solution).



(a) Scheduling steps (b) Workload fluctuations during scheduling

Figure 1: VM scheduling Phases

VMPP solutions stand and fall with their scalability, reliability and reactivity of properties, because they have to maintain a placement that satisfies the requirements of all VMs while optimizing the usage of CC resources. For instance, a naive implementation of a master/worker approach as described in Figure 1(a) would prevent workload fluctuations to be taken into account during the computation and the application of a schedule, potentially leading to artificial violations (*i.e.*, resource violations that are caused by the VMPP mechanism). In other words, the longer each phase, the higher the risk that the schedule may be outdated when it is computed or eventually applied, cf. the different loads during the three phases in Figure 1(b). Similarly, servers and network crashes can impede the detection and resolution of resource violations if the master node crashes or if a group of VMs is temporarily isolated from the master node.

VMPP solutions can only be reasonably evaluated if their behavior in the presence of such adverse events can be analyzed. Providing a framework that facilitates such studies and increases their reproducibility is the main objective of VMPlaceS.

3. 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 [15]) to allow users to investigate Cloud Computing challenges [19].
- 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 hosting more than 100,000 VMs on a computer with 16 GB of memory.

MS⇒AL Add a ref.

- 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 [11]).

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 [15]. These two capabilities are mandatory to build our VM placement simulator toolkit.

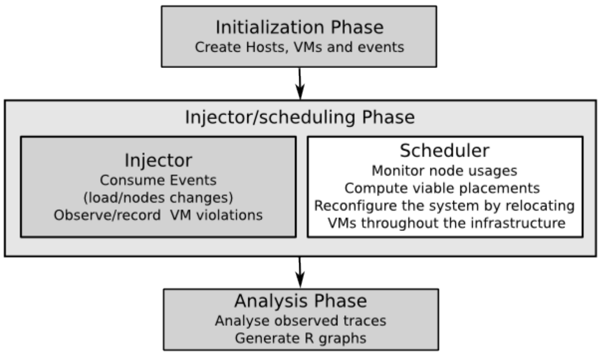

Initialization Phase
Create Hosts, VMs and events
Injector/scheduling Phase
Injector
Consume Events
(load/nodes changes)
Observe/record VM violations
Scheduler
Monitor node usages
Compute viable placements
Reconfigure the system by relocating VMs throughout the infrastructure
Analysis Phase
Analyse observed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

Figure 2: 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).

4. 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.

4.1 Overview

From a high-level view, VMPlaceS performs a simulation in three phases: (i) initialization (ii) injection and (iii) trace analysis (see Figure 2). 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. Currently, the supported events are VM CPU load change and node apparitions/removals that we use to simulate node crashes. It

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. 5, 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.

4.2 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 3, 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*.

4.3 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.

MS⇒AL maybe it is better to prevent the access to Xhost and XVM methods that can change the Simulator States. Hence, we should enforce the access only through the SimulatorManager class? What do you think? Yes, would be cleaner. Can we just present the interface as such? Or not talk about the direct possibility?

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.

4.4 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 [3], 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 [8] 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.

5. DYNAMIC VMPP ALGORITHMS

To illustrate the interest of VMPlaceS, we implemented three dynamic VM placement mechanisms respectively based on the Entropy [14], Snooze [12], and DVMS [22] proposals. For the three implementations, we chose to use the latest VMPP solver that has been developed as part of the Entropy framework [13].

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).

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

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.

5.1 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 4.4, 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.

5.2 Snooze-based Hierarchical Approach

We now present Snooze [12] 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 [12] and additional information stemming from personal communications of the Snooze developers and its implementation [5, 24].

5.2.1 Architecture

Snooze harnesses a hierarchical architecture in order to support load balancing and fault tolerance, cf. Fig. 3. 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.

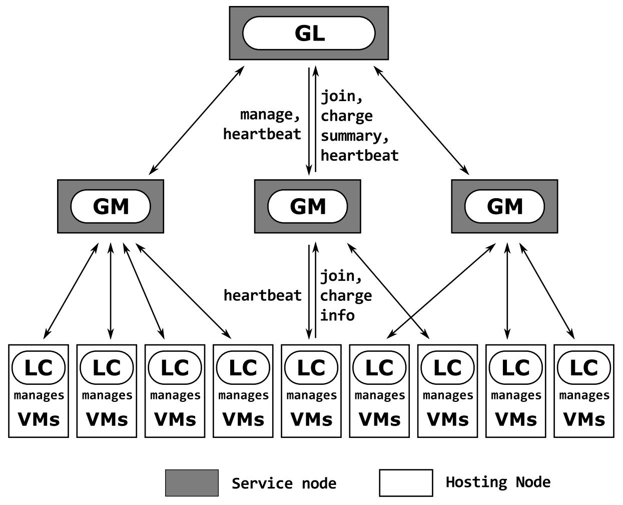


Figure 3: Overview of Snooze's architecture

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 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).

5.2.2 *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.

5.2.3 *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.

5.3 DVMS-based Distributed Approach

As the third use-case, we implemented the DVMS (Distributed Virtual Machine Scheduler) proposal [22]. Through a cooperative and fully distributed approach, DVMS aims at preserving resource requiements of VMs by migrating VMs located on overloaded servers to underloaded ones.

5.3.1 *Architecture*

A DVMS agent is deployed on each node composing the infrastructure. Agents are organized on top of an overlay network, which can be structured (such as Chord [23]) or unstructured. For the sake of this study, we implemented a simple but effective unstructured overlay that enables the simulated agents to collaborate without side effects: when it is requested, the overlay simply provide a link to a potential collaborator.

AL⇒JP How it is structured in the current situation

JP⇒AL I added the preceding paragraph, is it enough?

When a server is overloaded (i.e. VMs hosted on the server consume resources that can be produced), an *Iteraive Scheduling Procedure (ISP)* is started and a partition containing nodes considered for the reconfiguration will grow. Initially, the partition only contains the overloaded node.

Partitioning the infrastructure is mandatory to avoid conflicts between several schedulers that could manipulate the same nodes or VMs when they apply their reconfiguration plans.

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 last node of the partition: it will lead the scheduling computations, aiming at solving the overloading of the initiator;

In case of the scheduling computation is unsuccesful, a new node will be inserted in the partition, becoming the new leader of the partition.

5.3.2 *Iterative Scheduling Procedure*

AL⇒MS,JP if we succeed to have only one or two subsections in Snooze, we should do the same here.

When a node N_i detects that it cannot provide enough resources to its hosted VMs, it generates a partition and reserves itself to solve the problem (see Figure 4(a)) : the node N_i is the initiator of the partition. After that, the partition is forwarded to its closest neighbor N_{i+1}, given by the network overlay.

If N_{i+1} is already involved in another partition, it directly forwards the received partition to node N_{i+2}; otherwise, N_{i+1} joins the new partition (see Figure 4(b)) and checks that the scheduling procedure is still valid. If the partition is not valid anymore (for instance because VMs workload fluctuated), N_{i+1} cancels the reservations to destroy the partition and thus allow the nodes that composed it to take part to an other 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; in return, the new leader receives information regarding (i) the capacities of each node and (ii) the resources consumed by virtual machines that are hosted on each node. The leader then starts a scheduling computation; if no solution is found, the event is then forwarded to node N_{i+2}.

N_{i+2} repeats the same procedure: self-reservation (if it is free, see Figure 4(c)), event validity check, becoming the leader, monitoring of VMs and nodes inside the partition and launching a scheduling computation. If N_{i+2} finds a working reconfiguration plan, it is applied in order to solve the overloading of the initator; Once the application has been performed, the partition is dissolved and the nodes that composed it are now able to take part to other problem solving procedures.

Note that, if N_{i+2} did not find a solution, the partition would have grown until a solution was found or the partition would have used all the node constituting the infrastructure. In the latter case, the problem would be considered as unsolvable and the partition would be destroyed.

The progressing increase in size of the partition aims at adapting it to the complexity of the problem to solve. This approach enables to consider as few nodes as possible, thus accelerating the scheduling computations to solve the event as fast as possible.

5.3.3 *Fault-tolerance*

The main advantage of using overlay networks is that they have built-in fault tolerance mechanisms. Thus, to manage the connectivity between its nodes, DVMS works on top of an overlay network such as Chord: when a node needs to rebalance its VMs workload, it asks the overlay network to find some collaborators. In this study we implemented a simple overlay network as a flat list of agents : a typical request for collaborators includes a filter (i.e. a list of agents that is

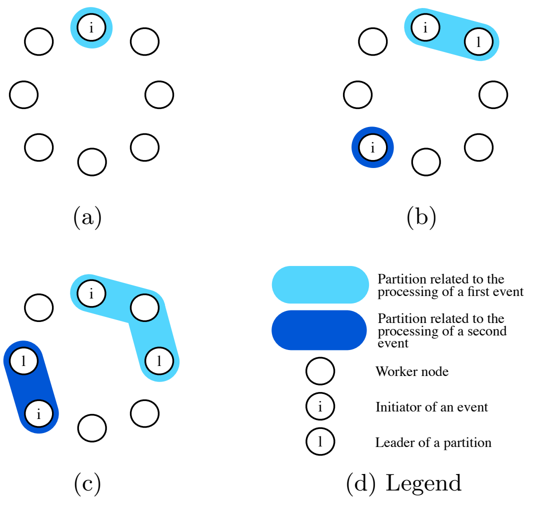

i
l
(a)
(b)
(c)
Partition related to the processing of a first event
Partition related to the processing of a second event
Worker node
Initiator of an event
Leader of a partition
(d) Legend

Figure 4: Processing two events simultaneously

already collaborating with the requesting agent), thus finding a suitable collaborator only consist in finding an agent that is not the filter. A link to the collaborator is then given to the requesting agent, which will only collaborate by the mean of message exchange containg immutable data: our implementation of the DVMS follows the guidelines advocated by the actor model, in order to ease the design of concurrent and distributed applications.

Even if the implementation of the overlay network is simple, it fulfills its purpose, and in the case where one would want to use a different overlay network such as a ring topology, it only has to reuse the API provided by this simple implementation, and adapts its functionning to the targeted overlay network.

AL⇒JP here we do not describe DVMS in general but we should emphasize what has been exactly implemented and how

JP⇒AL I added the preceding paragraph that gives more details about the overlay network used.

This functionning allows firstly a loosely coupling between DVMS and the overlay network used, and, secondly, to delegate most of the fault tolerance mechanisms to the overlay network. Although leveraging an overlay network to address node crashes is helpfull, it is not enough to make the problem solving procedure fault-tolerant.

Previously, if the leader of a partition crashed, the problem identified by the initiator would never be solved; moreover, the nodes of this partition would remain reserved indefinitely and would not be able to take part to other problem solving procedures.

To avoid these issues, DVMS now relies on a timeout. Each node involved in a partition periodically checks whether the state of its partition changed recently (for instance, a new node joined the partition). If the state does not change anymore, it probably means that the problem solving procedure is stuck (maybe because the leader crashed). In this case, each node decides to leave the partition and becomes free to take part to other problem solving procedures.

AL⇒JP, MS here we miss a paragraph that gives few details regarding how the overlay and the communication process have been implemented.

JP⇒AL I modified the first paragraph of this section.

6. EXPERIMENTS

AL⇒JP,AL,MS 2 pages

AL Il faudra parler du nombre de migrations qui est egalement une mÃl'trique pertinente. Plusieurs algorithms tentent de reduire cette metrique

AL⇒AL Il faudra mettre des snapshots de PajeNG

6.1 Comparison of Entropy, Snooze and DVMS

6.1.1 Reactivity and Violation time

6.1.2 Fault Tolerance

AL⇒JP perform an experiment on Snooze and DVMS with fault periodicity of 10 days for instance

MS⇒JP perform an experiment on Snooze with additional fautls (i.e. reducing the fault periodicity) in order to illustrate the pros/cons of the two different join mechanisms. To switch between one and the other strategies, see AUX.java line 34, GLElectionForEachNewGM = false/true

MS⇒JP

6.2 Investigating Variants and Possible Improvements

Our simulation framework facilitates the simulation of variants of placement algorithms. In the following, we present several variants of the placement algorithms introduced in Sec. 5 that have been discussed in the literature or come up during the implementation of their models using VMPlaceS. This section provides strong evidence that the modification and evaluation of existing algorithms is much facilitated by our simulation framework.

6.2.1 Variants of hierarchical scheduling

We now present three non-trivial variants that we have implemented and explored: periodic vs. reactive scheduling, a variant of the assignment algorithm of LCs to GMs, and a variant of the algorithms of how GMs and LCs join the system.

MS How and where do we provide the scheduling parameters for the evals.?

Periodic vs. reactive scheduling. Snooze [12] schedules VMs in a periodic fashion: after a fixed time period a GM calls the scheduler in order to resolve resource conflicts among the LCs it manages. The information whether a resource conflict has to be handled is taken based on the summary information that is periodically sent by the LCs to the GM.

We have implemented using VMPlaceS an alternative, reactive, strategy to scheduling: as soon as resource conflicts occur, LCs avert their GMs ofthem; the GMs then immediately initiate scheduling. Implementing this reactive scheme can be done using our framework in two manners. First, by implementing additional asynchronous transmissions of the necessary state updates as a real implementation would proceed. Second, in a much more lightweight manner through direct accesses by the GMs to the states of their respective LCs. In order to ensure that this leightweight implementation mimics a real implementation closely, delays induced by communication in the "real" implementation are accounted for (congestion issues are not relevant in this case because a resource conflict blocks the GM and its LCs anyway). We have implemented this lightweight variant of reactive scheduling including explicit modeling of communication delays. Using the abstractions provided by VMPlaceS, in particular harnessing its extended notion of hosts that represent the VMs managed by the LCs of a VM, reactive scheduling has been implemented by adding 4 lines of code.

We have shown the usefulness of our framework by exploring some properties of reactive scheduling compared to periodic scheduling. To this end we have simulated reactive scheduling and a periodic algorithm that reconfigures all LCs every 2 mins for configurations ranging from 64 to 512 LCs. These simulations have yielded, among others, the results shown in Tbl. 1. The results clearly show that while a reactive strategy entails a much higher number of migrations (because the periodic one misses many overload situations), reactive scheduling results in a significantly lower total migration time.

Assignment of LCs to GMs. LCs are assigned to GMs by the GL as part of the LC join protocol. In Snooze's native implementation LCs are assigned in a round-robin fashion to the known GMs. If GMs join (and leave) the system at the same time as LCs, a round-robin (RR) strategy at join time, however, does not ensure an even distribution. This may happen, for instance at startup time of the system, when new GMs and LCs enter the system, or in case of failures, which trigger GM and LC joins. In order to evaluate the corresponding imbalance and its consequences we have implemented the LC assignment protocol in a modular fashion and applied it to different highly-dynamic settings in which GMs and LCs enter the system at the same time. Furthermore, we have implemented a best-fit (BF) strategy that assigns LCs to the GMs with minimal load or, if several GMs with minimal load exist, to the GMs with the smallest number of assigned LCs.

We have evaluated the two LC assignment strategies using VMPlaceS on configurations of 128 and 256 nodes. In order to clearly expose the corresponding differences this evaluation has been performed by "stressing" the two strategies by simultaneously starting all LCs (128/256) and all GMs (12/25) and then simulating the resulting configuration over a one hour period. These experiments have yielded the fol-

Algorithm	No. migrations				Total violation time (s)			
#LCs/#GMs	64/6	128/12	256/25	512/51	64/6	128/12	256/25	512/51
Reactive	30	89	179	338	267	764	1638	3088
Periodic 120s	9	24	69	142	814	2548	6238	16038

Table 1: Reconfiguration nos. and violation times for reactive and periodic scheduling

Strategy	#LCs/#GMs				Total violation time (s)	
	128 LCs, 12 GMs		256 LCs, 25 GMs		128 LCs, 12 GMs	256 LCs, 25 GMs
	range	stdev	range	stdev		
Best-Fit	0–30	10.53	0–18	6.62	395	1005
Round-Robin	0–49	15.7	0–35	12.22	630	1265

Table 2: LCs to GM assignment and cumulated violation times for RR and BF strategies

lowing results, cf. Table 2:

- BF yields more homogeneous assignments of LCs to GMs: the ranges of the numbers of LCs assigned to a GM and their standard deviations are significantly smaller.[2]
- The cumulated time spent resolving violations is, for both configurations, significantly smaller for BF than for RR.

From these results, we can clearly infer that BF is significantly better than RR for the two tested kinds of configuration. Furthermore, we conjecture that BF should perform at least as good as RR for all configurations (the proof is left to future work).

Variants of the join algorithms. The join algorithms, see Sec. 5.2.2, are crucial to Snooze for two main reasons: (i) they have to be efficient because they can easily form a bottleneck if large numbers of LCs (GMs) have to be registered at a GM (LC); (ii) they are multi-phase protocols whose correctness especially in the presence of faults is difficult to ensure.

In order to investigate the corresponding trade-offs, we have used our framework to implement join algorithms that may be interrupted at any time, repeat the the on-going phase a number of times before reinitiating, if necessary, the entire protocol. Furthermore, the join protocol is parameterized, *e.g.*,, in the number of threads used to handle registration requests.

Finally, our framework has enabled us to test another aspect of Snooze's join algorithm as presented by Feller *et al.*. [12], a strategy we call the GM rejoin strategy (GRJ): all GMs and the LCs assigned to them should rejoin if a new GM enters the system. While GRJ supports a form of load balancing (because all LCs are reassigned to the new set of GMs), our simulation has shown that this strategy significantly increases the time necessary for registering GMs and LCs compared to a simpler strategy that does not modify existing GMs in case a new GM enters the system. This handicap is particularly pronounced if joins of GMs may be interrupted due to faults. Concretely, experiments involving 20 GMs and 200 LCs have shown that this strategy often multiplies the time necessary to join all 220 components by 10 or more compared to the simple join strategy. While the qualitative result that the more complex strategy presented in the paper results in a more time-consuming join process is not very surprising, the extent of the resulting degradation was surprising.

[2]Here, some GMs may be assigned 0 LCs if some GMs join the system after all LCs have been assigned.

6.2.2 DVMS Analysis

AL if space and time add PajeGN view for DVMS

7. RELATED WORK

AL⇒AL .25 page

Several simulator toolkits have been proposed since the last years in order to adress CC concerns [10, 17, 9, 21, 18]. 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 [10] 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 thus, they are not available for the scientific community. The second category [21, 18, 9] 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 [9], 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 [15].

8. CONCLUSION

AL⇒AL .25 page

Future work : (i) network changes and other dimensions (I/O), (ii) VM provisioning (i.e. Adding/removing VMs in the system,

9. ACKNOWLEDGMENT

This work is supported by the French ANR project SONGS (11-INFRA-13). Experiments have been performed 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).

AL Biblio try to fix only relevant publications in top ranked conferences

10. REFERENCES

[1] CloudStack, Open Source Cloud Computing. `http://cloudstack.apache.org`.

[2] Open Source Data Center Virtualization. `http://www.opennebula.org`.

[3] PajeNG - Trace Visualization Tool. `https://github.com/schnorr/pajeng`.

[4] Simgrid publications. `http://simgrid.gforge.inria.fr/Publications.html`.

[5] Snooze web site. `http://snooze.inria.fr`, Last access: 21 Oct. 2014.

[6] The Open Source, Open Standards Cloud. `http://www.openstack.org`.

[7] A. Barker, B. Varghese, J. S. Ward, and I. Sommerville. Academic Cloud Computing Research: Five Pitfalls and Five Opportunities. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, June 2014.

[8] V. A. Bloomfield. *Using R for Numerical Analysis in Science and Engineering*. Chapman & Hall/CRC, 2014.

[9] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. 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.

[10] J. Cartlidge and D. Cliff. 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*, pages 58–68, 2013.

[11] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, scalable, and accurate simulation of distributed applications and platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[12] E. Feller, L. Rilling, and C. Morin. Snooze: A Scalable and Autonomic Virtual Machine Management Framework for Private Clouds. In *CCGRID '12: 12th Int. Symp. on Cluster, Cloud and Grid Comp.*, pages 482–489, May 2012.

[13] F. Hermenier, S. Demassey, and X. Lorca. Bin Repacking Scheduling in Virtualized Datacenters. In *CP '11: Proceedings of the 17th international conference on Principles and practice of constraint programming*, pages 27–41. Springer, 2011.

[14] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM.

[15] T. Hirofuchi, A. Lebre, and L. Pouilloux. 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*, CLOUDCOM '13, pages 96–103, Washington, DC, USA, 2013. IEEE Computer Society.

[16] J. Hu, J. Gu, G. Sun, and T. Zhao. A scheduling strategy on load balancing of virtual machine resources in cloud computing environment. In *Parallel Architectures, Algorithms and Programming (PAAP), 2010 Third International Symposium on*, pages 89–96, Dec 2010.

[17] A. Iosup, O. Sonmez, and D. Epema. Dgsim: Comparing grid resource management architectures through trace-based simulation. In E. Luque, T. Margalef, and D. Benítez, editors, *Euro-Par 2008 – Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin Heidelberg, 2008.

[18] D. Kliazovich, P. Bouvry, Y. Audzevich, and S. Khan. Greencloud: A packet-level simulator of energy-aware cloud computing data centers. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5, Dec 2010.

[19] J. L. Lucas-Simarro, R. Moreno-Vozmediano, F. Desprez, and J. Rouzaud-Cornabas. Image transfer and storage cost aware brokering strategies for multiple clouds. In *IEEE CLOUD 2014. 7th IEEE International Conference on Cloud Computing*, Anchorage, USA, June 27-July 2 2014. IEEE Computer Society.

[20] R. Moreno-Vozmediano, R. Montero, and I. Llorente. IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures. *Computer*, 45(12):65–72, 2012.

[21] A. Nunez, J. Vazquez-Poletti, A. Caminero, G. Castané, J. Carretero, and I. Llorente. icancloud:

A flexible and scalable cloud infrastructure simulator. *Journal of Grid Computing*, 10(1):185–209, 2012.

[22] F. Quesnel, A. Lebre, and M. Südholt. Cooperative and Reactive Scheduling in Large-Scale Virtualized Platforms with DVMS. *Concurrency and Computation: Practice and Experience*, 25(12):1643–1655, Aug. 2013.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. 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*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[24] S. D. Team. Snooze characteristics and implementation, July 2014. Personal communication.

[25] H. N. Van, F. Tran, and J.-M. Menaud. Sla-aware virtual resource management for cloud infrastructures. In *Computer and Information Technology, 2009. CIT '09. Ninth IEEE International Conference on*, volume 1, pages 357–362, Oct 2009.

[26] M. Wang, X. Meng, and L. Zhang. Consolidating virtual machines with dynamic bandwidth demand in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 71–75, April 2011.