Snooze: A Scalable, Fault-Tolerant and Distributed Consolidation Manager for Large-Scale Clusters

Eugen Feller*, Louis Rilling†, Christine Morin*, Renaud Lottiaux†, Daniel Leprince‡
*INRIA Centre Rennes - Bretagne Atlantique
Campus universitaire de Beaulieu, 35042 Rennes Cedex, France
{Eugen.Feller, Christine.Morin}@inria.fr
†Kerlabs
80 avenue des buttes de Coësmes, Bâtiment Germanium, 35700 Rennes, France
{Louis.Rilling, Renaud.Lottiaux}@kerlabs.com
‡Électricité de France (EDF) R&D
1 avenue du Général de Gaulle, 92140 Clamart, France
Daniel.Leprince@edf.fr

***Abstract*—Intelligent workload consolidation and dynamic cluster adaptation offer a great opportunity for energy savings in current large-scale clusters. Because of the heterogeneous nature of these environments, scalable, fault-tolerant and distributed consolidation managers are necessary in order to efficiently manage their workload and thus conserve energy and reduce the operating costs. However, most of the consolidation managers available nowadays do not fulfill these requirements. Hence, they are mostly *centralized* and solely designed to be operated in *virtualized environments*.**

In this work, we present the architecture of a novel scalable, fault-tolerant and distributed consolidation manager called *Snooze* that is able to dynamically consolidate the workload of a software and hardware heterogeneous large-scale cluster composed out of resources using the virtualization and Single System Image (SSI) technologies. Therefore, a common cluster monitoring and management API is introduced, which provides a uniform and transparent access to the features of the underlying platforms. Our architecture is open to support any future technologies and can be easily extended with monitoring metrics and algorithms. Finally, a comprehensive use case study demonstrates the feasibility of our approach to manage the energy consumption of a large-scale cluster.

***Keywords*-Energy Management, Cluster, Virtualization, SSI, Consolidation, Heterogeneity, Scalability, Dynamic Adaptation**

I. Introduction

Energy management for mobile devices has been traditionally a well studied topic during the last two decades, as these devices usually do not have a permanent connection to the power grid and thus solely rely on the limited battery charge. However, this trend has been mostly disregarded in the context of HPC systems as the main focus mainly relied on improving the performance at any cost. Therefore, energy costs for operating and cooling the equipment of current data centers have increased significantly up to a point where they are able to surpass the hardware acquisition costs. Studies have shown that data centers alone have consumed 61 billion kWh of U.S. energy in 2006. This is enough energy to power 5,8 million average U.S households and results in approximately $4.5 billion/year of energy costs [1]. These numbers are most likely to increase up to 120 billion kWh by 2011 in case no further energy conservation steps are taken [1]. Not least, the way energy is generated influences our environment either directly by the carbon footprint or indirectly by the nuclear waste. According to the Environment Protection Agency (EPA) decreasing the energy consumption could reduce these wastes by 15 to 47 million metric tons in 2011 [1].

Reducing the energy consumption requires to understand where most of the energy is spent. Server hardware is typically over-provisioned in order to sustain the service availability during periods of peak demand. However, resource demand in current data centers is usually of a bursty nature and thus results in a low average utilization of approximately 15-20% [2]. Therefore, a big fraction of the resources can be used to take energy conservation decisions such as suspending or turning off unnecessary servers, while still preserving the performance requirements. Given that ubiquitous virtualization and SSI solutions are able to *migrate* the workload and servers can be *turned on and off* at any time, clusters can be dynamically adapted depending on the resource demands. Consolidation of virtual machines on a subset of physical nodes is a well known technique to reduce the number of active physical resources and has been studied in several works. In [3], a consolidation manager called Entropy is introduced, which dynamically maps the virtual machines to the available resources. Thereby, in order to achieve task isolation each task is usually assigned to a VM, which is then taken under control of a Virtual Machine Monitor (VMM) such as Xen [4] or KVM [5]. Similar examples for consolidation managers can be found in [6], [7] and [8]. However, all these solutions are *highly centralized* and do not take into account the *software heterogeneous* nature of a cluster, where nodes can be either virtualized or part of a cluster running an SSI operating system such as Kerrighed [9]. Kerrighed provides the user with the illusion that a cluster is a big SMP machine. Similarly to the virtualization approach, workload can be migrated among the cluster. However, it is fine-grained and thus represents a single task.

In this paper, we present Snooze, a scalable, fault-tolerant and distributed energy-performance aware consolidation manager for software and hardware heterogeneous large-scale clusters. Therefore, in order to bridge the gap between the different calling semantics of the underlying techniques (i.e. virtualization and SSI) we introduce a so called *Common Cluster Monitoring and Management API* as a uniform interface to transparently monitor and manage these systems. Thereby, our solution is not bound to any specific technique and can be used to manage any existing heterogeneous cluster setup. Furthermore, its *hierarchical* architecture, *fault-tolerance with replication* and a *dedicated overlay network* for the framework components make Snooze decentralized, scalable and fault-tolerant.

The remainder of this paper is organized as follows. Section II presents the theoretical foundations of our work. Section III details the architecture, its components and their interactions. Section IV provides a use case study. Section V discusses the related work. Finally, Section VI closes the paper with conclusions and future work.

II. Theoretical Foundations

The main objective of our work is to minimize the number of machines hosting the workload. Therefore, we divide this paragraph into two parts: consolidation management and idle-time management. In the first part, we first provide a formal definition of the workload placement problem and then present an algorithm to approximate a solution. In the second part, we focus on idle-time management and detail how we intend to determine the idle-time threshold which needs to be reached in order to achieve energy savings and finally predict the idle periods to suspend or turn off idle machines.

A. Consolidation Management

1) Formal Problem Definition: We define the problem of mapping the workload to physical machines as an instance of a one-dimensional bin-packing problem, in which the physical machines represent the bins and workload the items to be packed. Each bin has predefined resource capacity and all items are assigned with a resource demand, the so called dimension. Thereby, currently only one dimension is taken into account and represents the CPU demand. In the following we introduce the notations and provide a formal definition of this problem.

Let B denote the set of bins and I the set of items, with $n = |B|$ and $m = |I|$ representing the amounts of bins and items. Furthermore, each bin and item is assigned with a predefined resource capacity C_j and demand c_i respectively. In addition, we define the following two binary decision variables:

1) Bin allocation variable y_j, equals 1 if the bin $j \in B$ is chosen, and 0 otherwise.
2) Item allocation variable $x_{i,j}$, equals 1 if the item $i \in I$ is assigned to the bin $j \in B$, and 0 otherwise.

The ultimate goal of the consolidation algorithm is then to place all items such that, the number of bins used is minimized. This is reflected in our objective function (1).

$$\text{Minimize} \sum_{j=1}^{n} y_j \tag{1}$$

Subject to the following constraints:

$$\sum_{i=1}^{m} c_i x_{i,j} \leq C_j y_j, \forall j \in B \tag{2}$$

$$\sum_{j=1}^{n} x_{i,j} = 1, \forall i \in I \tag{3}$$

The constraint (2) ensures that the capacity of each bin is not exceeded and constraint (3) guarantees that each item is assigned to exactly one bin.

2) Solution Methodology: The problem defined in the previous paragraph has been widely studied in the literature and shown to be NP-hard [10]. Therefore, approximation solutions are necessary in order to find reasonable results in acceptable run-time. We use a heuristic approach to solve this problem. However, one of the major drawbacks of current heuristic algorithms (e.g. FFD) for bin-packing problems is that they are static. Hence, the objects are packed once and are not allowed to be taken out of the bins. Thereby, bin capacity is often wasted. Nevertheless, the migration functionality provided through virtualization and SSI technology allows us to move the workload. Thus, heuristic algorithms with relaxed constraints (i.e. migration) are needed to increase the bin capacity usage.

In the following, we present an on-line heuristic called better-fit, initially introduced in [11] and used in our work to approximate a solution. Better-fit makes use of the migration functionality to optimize the bin capacity usage and thus is well suited for our work. It assumes that the arrival of the workload is sequential and works as follows. Each time a new workload arrives the nodes and the corresponding workloads are inspected, starting from the first node. Thereby, the existing workload resource demands (i.e. CPU) are compared with the arriving workload resource requirements. In case the arriving workload is able to fill the node better than some existing one, it is inserted and the replaced workload is assigned again into another bin using the best-fit heuristic [11]. This heuristic then inserts the old workload into a node which has the smallest room to accept it.

A similar procedure happens when some workload finishes. In that case, it might be possible to consolidate the left over workload among the cluster, in order to suspend or turn off a machine. Therefore, workload removal operation evicts the current workload running on a node and uses the insert operation to search for a new allocation.

We use the following two figures to describe the addition and removal operations of this algorithm. In Figure 1 the insert operation is illustrated based on two nodes.

The first node runs a workload which amounts to 90% of the total node capacity. When new workload *WL8* arrives it needs to be placed on a machine. The algorithms starts from the

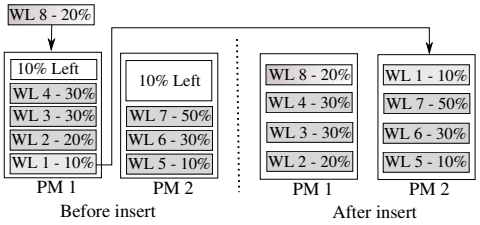

Fig. 1. Workload addition operation

first node and evicts WL 1 as this node can be better utilized by replacing *WL 1* with *WL 8*. Finally, *WL 1* is migrated to the second machine, as it has 10% of free capacity. On the other hand, if it did not have enough free capacity left a new machine would be turned on.

Figure 2 shows the removal operation. Similarly to the previous example we use two nodes. The former node has 10% of spare capacity, while the latter one is fully loaded. When *WL 6* terminates, all the current workload (i.e. *WL 5*) is evicted and placed on the first machine using the insert operation. Therefore, idle period is created, giving the opportunity to suspend or turn off the machine. However, transitioning the idle machine into a lower power-state does not necessarily yield to energy savings. We will detail how to determine idle periods with energy gains in paragraph II-B.

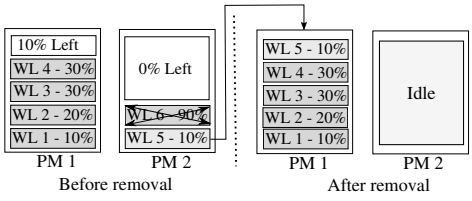

Fig. 2. Workload removal operation

Finally, workload resource requirements are of dynamic nature. Therefore, underutilization is most likely to happen on any node after the initial allocation. Furthermore, existing workload is able to change its current resource requirements and thus overload a machine, leading to performance degradations of co-existing workload. This can either happen as a result of a VM resize event within a virtualized server or increased resource (e.g. CPU) usage on a non-virtualized time-shared SSI machine. Both cases need to be avoided as they can result in energy wasting and lead to performance degradations of existing workload. Thus, reconfiguration actions need to be taken upon resource usage decrease and increase. In the former case underutilization is detected and all the workload currently running on a machine is remapped using the insert operation. In the later case the *hot workload* is similarly remapped by invoking the insert operation. Figure 3 shows the process of workload consolidation upon underutilization detection. When the resource utilization of a machine is low (e.g. 20% at PM2) all its workload (i.e. WL 3 and 4) is remapped using the insert operation. Thereby, in order to limit the number of reconfigurations caused by resource usage variability, we define a settling time which needs to elapse before a new reconfiguration can be triggered.

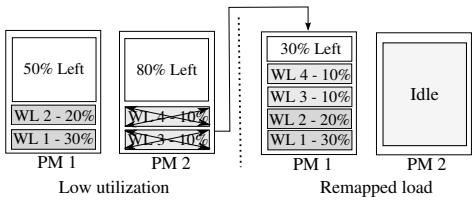

Fig. 3. Resource underutilization

B. Idle-time Management

After the consolidation, idle periods are created and servers should be transitioned into a lower power-state (e.g. suspend), in order to maximize the energy savings [12]. However, suspending the system does not necessarily yield into any energy saving if the idle periods are not long enough, as every state transition consumes additional energy and introduces computation delays. Thus, if done too frequently any potential energy savings can be destroyed. Therefore, it is necessary to determine the *idle-time threshold* which needs to be reached to achieve some energy gains. Consequently, if the idle period is below this threshold, it is advantageous to keep the system running. Therefore, knowing the idle periods in advance is necessary in order to take energy conservations decisions. Unfortunately, this information is usually not available and needs to be estimated.

In the following paragraph we detail a simple, yet efficient approach on how to determine the *idle-time threshold* which yields to energy saving and *estimate* the *idle periods*.

1) Calculating the idle-time threshold: We first define T_i as the idle period. Moreover, we define T_d as the delay overhead to enter the suspend state, T_s as the time in the suspend state and T_w as the delay overhead to resume from the suspend state (see Figure 4). Note, that T_s can be easily computed as the difference between T_i and T_d.

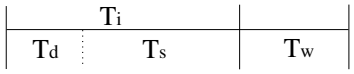

Fig. 4. Idle-period definition

In addition, we define P_i as the idle power consumption and P_s as the power consumption in the suspend mode. Last but not least, we define P_{sw} as the average-power consumption for entering and resuming the suspend state. Assumed that the idle period is longer than the delay overhead to enter the suspend state, the system will always enter the suspend state. In order to calculate the potential energy conserved we define the energy profit EP by entering the suspend state as follows:

$$EP = P_i \times T_i - (T_d + T_w) \times P_{sw} - P_s \times T_s > 0 \quad (4)$$

$$= P_i \times T_i - (T_d + T_w) \times P_{sw} - P_s \times (T_i - T_d) > 0 \quad (5)$$

$$= T_i \times (P_i - P_s) - T_d \times (P_{sw} - P_s) - P_{sw} \times T_w > 0 \quad (6)$$

The threshold which yields to energy savings is then given by:

$$T_i > \frac{T_d \times (P_{sw} - P_s) + P_{sw} \times T_w}{(P_i - P_s)} \quad (7)$$

$$\Rightarrow T_{threshold} = \frac{T_d \times (P_{sw} - P_s) + P_{sw} \times T_w}{(P_i - P_s)} \quad (8)$$

2) Idle-period prediction: As observed in the previous paragraph, T_i parameter remains to be the unknown and needs to be estimated in order to be able to suspend the system in advance. We use the well known exponential moving average equation to predict the idle periods. Therefore, we monitor and keep track of the previous idle times. Afterwards, we accumulate these values to predict the upcoming idle period using the following recursive equation:

$$T_{i+1} = c \times t_i + (1 - c) \times T_i \quad (9)$$

where T_{i+1} is the newly predicted idle period, t_i is the most recent idle period, T_i is the last predicted value and c a constant in the range between 0 and 1. With this constant we can define either more weight should be given to the most recent idle period or to the previously predicted idle periods. Therefore, a value of 0.5 can be chosen to assign equal weight to all idle periods. In fact, it can be tuned according to the type of workload.

Finally, we transition the nodes into a lower power-state when $T_{i+1} > T_{threshold}$ holds.

III. System Architecture

This section details the architecture of Snooze, a scalable and fault-tolerant energy-aware consolidation manager for software and hardware heterogeneous large-scale clusters. Thereby, several properties have to be fulfilled by Snooze in order to adapt such an environment. First, it has to scale across many thousands of nodes. Second, nodes and thus framework management components can fail at any time. Therefore, the system needs to self-heal and continue its operation despite of component failures. Third, it needs to be able to adapt to a software and hardware heterogeneous cluster environment composed out of virtualized and non-virtualized SSI machines.

In order to obtain the first property, Snooze uses a *hierarchical* and *decentralized* architecture, which allows it to scale with the number of nodes. The second property (i.e. fault-tolerance) is achieved with *replication* and a *dedicated overlay network* for the framework components. Finally, software and hardware heterogeneity is assured by introducing a *Common Cluster Monitoring and Management API*, which provides a uniform and transparent access to the features (e.g. monitoring and workload control) of the underlying techniques (i.e. virtualization and SSI).

In the following paragraphs, we first introduce our system model and describe its assumptions. Afterwards, we give a global overview of the framework, detail its components and their interactions.

A. System Model and Assumptions

Our work targets heterogeneous large-scale clusters whose nodes are interconnected with a high-speed LAN connection such as Gigabit Ethernet or Infiniband. Furthermore, each node can be managed by any virtualization solution (e.g. Xen [4], KVM [5], OpenVZ [13], etc.) or an SSI operating system [9]. Therefore, in order to define workload in these two cases we introduce the notion of an *Application* and *Application Component*. Each application aggregates one or multiple application components, with each component representing a process tree, running either inside a VM or on top of an SSI node. Thereby, multiple components could co-exist on the same VM and SSI node as long as enough resource capacity (e.g. CPU, RAM, etc.) is available. However, the platform boundaries need to be respected. Thus, it is not possible to migrate a process running on the SSI cluster to a virtualized environment. Finally, we do not impose any restrictions on the type of the components. Hence, both service and computing applications are supported.

Figure 5 illustrates the mapping of the workload (i.e. VMs and application components) to physical machines. Here, we distinguish between two cases: virtualized and non-virtualized. In the former case, Snooze assumes that the application components are already mapped to VMs, and assigns the *VMs to physical machines*. In the latter case the application components are executed directly on top of the SSI nodes. Thereby, Snooze is in charge of mapping the *application components to physical machines*.

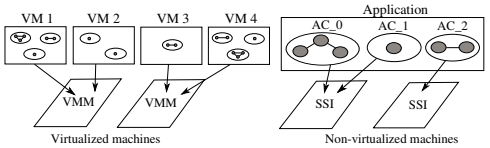

Fig. 5. Workload mapping

Thereby, we assume that the workload and its description (i.e. resource requirements, SLAs, etc.) are hosted on a distributed file system (e.g. XtreemFS [14]) and available to all machines in the cluster.

B. Global Overview

The layered architecture overview of Snooze is shown in Figure 6. It is composed out of three layers: *physical*, *overlay* and *client*. At physical layer, machines are organized within a software and hardware heterogeneous cluster, in which each node is controlled by the use of a so called *Local Controller*. In addition, an hierarchical overlay layer exists in order to efficiently manage the cluster. The overlay layer is composed out of fault-tolerant components: *Group Managers* and a *Group Leader*, which are organized within an overlay network. Thereby, each group manager manages only a subset

of nodes of the physical layer. Furthermore, a group leader exists and keeps a summary of the group managers. Finally, a client layer is used to provide an interface to the outside world. It is implemented by a predefined number of replicated *Entry Points*, and provides the functionalities for new nodes and group managers to join the overlay network. Thereby, a client can be any entity (e.g. cloud infrastructure, application manager, web portal, console, etc.) which uses the provided bindings to manage its workload. A binding is used to access the features of the *client interface* implemented by the entry points, such as:

- Resource monitoring: CPU, RAM, power, temperature, network latency and throughput
- Workload control: submission (including QoS requirements and workload priority), removal, resizing

In the rest of this paragraph we describe the details of each layer.

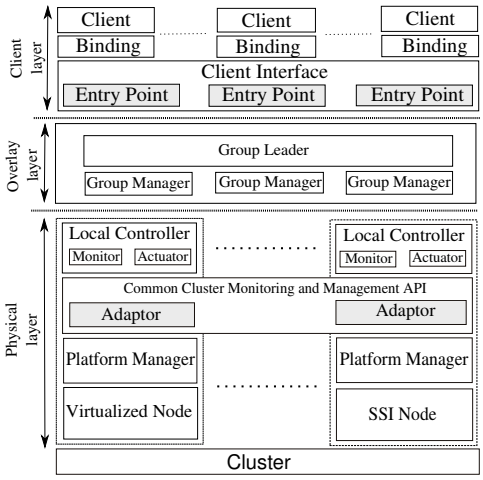

Fig. 6. Layered architecture overview

At the physical layer, there exists one *Platform Manager (PM)* per node, which is in charge of managing the resource. Therefore, it implements all the necessary functionality to monitor and control the physical machine. In case of a virtualized server, existing PM such as *libvirt* [15] is available and provides a uniform interface to most of the currently available virtualization solutions. On the other hand, non-virtualized servers, which can be part of an SSI cluster (e.g. Kerrighed [9]) have different calling semantics and thus are typically managed by a distinct PM (e.g. libkerrighed [16]).

In order to bridge the gap between the different calling semantics of the PMs we introduce the so-called *Common Cluster Monitoring and Management API*. This API provides a uniform interface to access the functionalities of the underlying PMs. It is implemented by a dedicated *Adapter* and used by the local controller to interact with the machine. The following functionalities are currently defined by this API:

- Monitoring: CPU, RAM, power, temperature, network latency and throughput
- Workload control: start, stop, migrate, resize
- Resource control: suspend, hibernate, node on/off, dynamic voltage and frequency scaling (DVFS)

Each node runs a *Local Controller*, which transparently loads the corresponding PM, implements the functionality to join the overlay, monitors the node resource usage and sends keep-alive messages to the assigned group manager. Furthermore, it is in charge of executing the management commands (e.g. start, stop, migrate, node on/off, etc.) coming from the group manager. Hence, it is composed out of two components: *Monitor* and *Actuator*. The former component implements *Probes* and thus monitors various resource usage metrics (e.g. CPU, RAM, power, temperature, network latency and throughput, etc.) of the system. We distinguish between two types of probes: active and passive. Active probes are periodically woken up by a timer and send their information to the assigned group manager within the overlay, while passive ones are invoked by the group manager directly via polling. Finally, the *Actuator* component is used to execute the workload and resource control requests (i.e. start, stop, migrate, resize, suspend, hibernate, node on/off, DVFS, etc.) coming from the group manager.

The overlay layer is depicted in Figure 7. It has a hierarchical structure and is composed out of two main components: *Group Managers and Group Leader*.

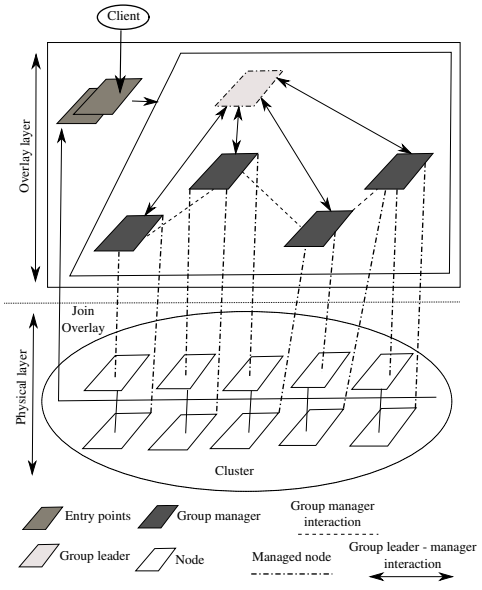

Fig. 7. Hierarchical architecture overview

Each node of the physical layer is assigned to one group manager within the overlay. Furthermore, there exists a set of group managers, where each group manager is responsible for a subset of nodes. Thereby, node management is achieved within a closed-loop by monitoring the resources, estimating the workload resource demands based on history data and applying an instance of a workload consolidation algorithm such as introduced in Section II. However, applying the consolidation algorithm on a subset of nodes does not lead to a global optimal solution. Hence, a group leader exists in order

to facilitate global decisions. Therefore, it maintains a summarized global view of the group managers and implements a global policy which is able to detect *hot group managers* and trigger reconfiguration actions. The summary only includes the current status (e.g. online, offline, etc.) of the group managers and the sum of the resource utilizations of the managed nodes. Therefore, network communication overhead is reduced, while still having enough information to facilitate global decisions (see Section IV).

The overlay join process of a group manager works as follows. Each time a new group manager attempts to join the overlay it sends a message to one of the entry points. The entry point then does a group leader lookup in its local cache. In case a valid group leader was found the joining group manager is taken under control and starts sending keep-alive messages to the group leader. Otherwise, a multicast message is sent into the overlay in order to discover the current group leader. Thereby, only the overlay participants (i.e. group managers and group leader) are affected. Finally, if no group leader exists the new group manager becomes the current group leader.

Given that the group leader can fail at any time, failure detection is performed by each group manager. This is achieved using the keep-alive messages. If one group leader fails, the keep-alive messages are lost and the process of new group leader election is started among the group managers. Therefore, each group manager sends a multicast message with the current resource utilization into the overlay. In doing so, the group member with the less utilized resources (e.g. CPU, RAM, network bandwidth, etc.) becomes the new group leader. Thereby, consensus among the group managers is achieved by implementing one of the existing leader election algorithms (e.g. [17]). Finally, the elected group leader redistributes its currently controlled resources (i.e. nodes) among other group managers, and requests the other group managers to join again. Consequently, the group managers start sending their resource monitoring summaries and keep alive messages to the new group leader and thus the summarized global view is rebuilt.

The overlay join process of a node is similar to the one of the group manager. First, a request is sent to one of the entry points by the local controller, which is then forwarded to the group leader. The group leader then assigns the new node to one of the managers according to the summarized utilization view and the deployed global policy (e.g. assign a node to the group manager whose nodes are undergoing excessive resource demands). Afterwards, keep-alive messages and monitoring information is sent to the group manager periodically and kept within a local database for each node managed by a group manager. In case of a group manager failure, keep-alive messages are lost and the process of a rejoin is started by the managed nodes in the same manner as the traditional join. Finally, if a managed node fails the node status information within the corresponding group manager is updated.

IV. Use Cases

We distinguish between two use cases. The first one details the process of workload submission, removal and resizing. The second one demonstrates the ability of Snooze to detect and react to local anomalies such as: *thermal emergencies (i.e. overheating)*, *underutilization* and *overload situations*.

A. Case 1: Submission, removal and resizing of the workload

When a user submits a request to start new workload to one of the entry points, a group leader lookup is done in the entry point cache. In case a valid group leader was found the request is forwarded to it directly. Otherwise, a multicast message is sent into the overlay in order to discover the current group leader. The group leader then uses the summarized resource utilization of the group managers to redirect the request to a group manager which has spare capacity left. The workload is then mapped to one of the physical machines managed by the group manager using the integrated consolidation algorithm. Similarly, when a message to terminate or resize a workload is received by the entry point, it is forwarded to the current group leader. The group leader then forwards the message to the corresponding group manager within the overlay. Finally, the group manager executes the operation and sends a reply message back to the group leader.

B. Case 2: Ability to detect and react to local anomalies

In order to detect local underutilization, overheating, and other events, monitoring information of the managed nodes is stored by the corresponding group manager and analyzed within a Monitor-Estimate-Plan-Execute (MEPE) loop. In case of resource underutilization a predefined threshold exists. When the system load on one of the managed nodes falls below this threshold, reconfiguration actions are triggered by the corresponding group manager. Therefore, the utilization of the workload is estimated using the recorded history values, consolidation algorithm is executed and the workload is migrated to nodes having enough capacity to host it. Thereby, as each group manager only manages a subset of nodes, the algorithm is not aware of possible resources available on other group managers. In fact, this is not necessary as it introduces additional communication overhead to obtain this information. Hence, the algorithm first tries to place the workload on the managed nodes. If there is no spare capacity left a message is sent to the group leader, in order to find a group manager which is able to host the workload. The group leader uses the summary information to find a proper target group manager and returns its location to the initiating group manager. The initiating group manager then issues a reconfiguration request to the target group manager in order to find a proper mapping of the workload among his managed resources. Afterwards, the workload is migrated to the allocated resources by the target group manager.

A similar procedure is executed when some resource becomes *hot* (i.e. high temperature). Depending on the policy enabled on the group manager, workload is evicted and placed on another resource by first trying to reallocate it within the local set of managed nodes, and finally by contacting the group leader. Alternatively, a policy could also reduce the

temperature by scaling down the processor frequency (i.e. DVFS).

V. Related Work

With the recent advances in virtualization technology, efficient workload consolidation has recently gained a lot of research interest. As a result several consolidation algorithms (e.g. [18], [19], [20], etc.) and energy-aware resource management frameworks (e.g. [6], [8], [3], [7], [21], [22], etc.) have been proposed.

In [6], a framework called *pMapper* is introduced. Therefore, the authors present an architecture and several model-based workload consolidation algorithms which are validated by using server utilization traces. However, their model-based workload consolidation algorithms rely on a number of assumptions which can not be fulfilled in real environments (e.g. migration costs are independent of the load within the VM). In addition, their architecture is highly centralized, limited to virtualized environments and does not tolerate component failures.

In [8], the *EnaCloud* framework is presented. Similarly to our work the authors use a heuristic algorithm to determine an energy-saving application placement. Nevertheless, the authors do not detail how they intend to manage the resulting idle periods. Moreover, their architecture relies on a centralized global controller, and thus can not scale to be used within a large-scale data center.

In [3], a consolidation manager called *Entropy* is introduced. Thereby, a constraint programming approach is used to find a mapping of VMs to physical machines. However, a centralized global decision module is used to adapt a virtualized environment according to the current resource usage.

In [7], a centralized energy-aware framework called *VirtualPower* is presented. Thereby, the notion of local and global polices is introduced in order to optimize the energy-consumption at node and cluster level. Our work also defines the notion of local and global policies. However, our local policies are used to manage a *subset* of nodes, while the global ones are used to facilitate a global solution.

The most related work in terms of architecture can be found in [21] and [22]. In [21], the authors introduce several heuristics for workload consolidation and validate them within a simulator. Moreover, a brief overview of a decentralized system architecture is given. However, the architecture still relies on a centralized non-fault-tolerant dispatcher, expensive exchange of resource and workload utilization among the global managers, virtualized environment and a no further defined distributed heuristic algorithm. Finally, as the main focus of this work was rather on simulations than on the framework, no further details about the architecture were specified.

In [22], a hierarchical cloud management system called *Eucalyptus* is introduced. We could have designed Snooze as an extension of an existing cloud management software like Eucalyptus. However, we believe that it is easy to provide a generic implementation of Snooze for any cloud manager. In particular, Snooze mostly implements a new resource management policy, that uses a simple, fault-tolerant, and scalable run-time. This run-time is kept simple and light-weight, and does not aim at replacing the base cloud management software itself.

In contrast to all these works, Snooze does not rely on a single instance which executes the consolidation algorithm. The *group managers* in Snooze are fully decentralized with each of them managing a subset of nodes and applying an instance of a centralized workload consolidation algorithm. Therefore, no distributed algorithms are needed and no monitoring information needs to be exchanged among them. Furthermore, our *group leader* is fault-tolerant and does not need to maintain a detailed global view of the cluster. In fact, it only keeps a summary to facilitate global decisions. Finally, by utilizing the generic cluster monitoring and management API, Snooze is able to manage any software and hardware heterogeneous cluster composed out of virtualized and non-virtualized machines.

VI. Conclusions and Future Work

In this paper, we have presented *Snooze*, an architecture of a novel scalable, fault-tolerant and distributed consolidation manager for heterogeneous clusters. To the best of our knowledge, this is the first work towards providing a decentralized framework for energy-management of heterogeneous large-scale clusters. By utilizing the common cluster management and monitoring API, Snooze is able to integrate and adapt any underlying *software and hardware heterogeneous* cluster setup, composed out of virtualized and non-virtualized machines running an SSI operating system in a uniform and transparent way. Furthermore, its *hierarchical architecture* with *replication and a dedicated overlay network for group managers* make it scalable, decentralized, fault-tolerant and thus suitable for managing the energy consumption of large-scale clusters, such as found in data centers of current Cloud providers. In fact, using the client bindings it is able to manage any cluster environment.

Moreover, we have formulated the problem of energy-efficient workload placement and used a heuristic algorithm to demonstrate one application of the framework. In addition, we have detailed our approach of idle-time management and used it to determine when it is worthwhile to automatically suspend or turn-off idle servers. However, our framework is not limited to any particular algorithm and can be easily extended with others (e.g. genetic algorithms). Finally, the versatility of Snooze can be used to implement any other cluster management policy such as: power capping using DVFS, migration according to workload QoS requirements such as network latency and throughput, etc.

Currently, we are implementing a first prototype of the framework, which will be validated within a heterogeneous cluster environment in the context of Grid5000 [23]. In addition, we are investigating the impact of consolidation on energy and performance for different types of workloads (i.e. service and computing). In fact, for now only one resource

component (i.e. CPU) is taken into account while performing the consolidation. Thus, workload characteristics such as memory usage and I/O patterns are ignored. We are aware that ignoring workload patterns could lead to performance degradations and result in increased energy consumption. Therefore, we plan to work on designing energy-aware workload consolidation algorithms taking into account workload patterns (e.g. consolidate workload with distinct resource usage characteristics) and QoS requirements. Moreover, migration is typically a costly operation in terms of time and energy and needs to be avoided as much as possible. Thus, we are currently investigating in designing algorithms which will minimize the number of migrations. In addition, even if exponential moving average is able to estimate the idle periods very efficiently, it still has drawbacks and tends to fail predicting sudden long idle periods. Therefore, we plan to investigate in more advanced prediction techniques. Last but not least, data centers of current Cloud providers typically aggregate multiple clusters. The hierarchical architecture of Snooze could be easily extended to manage these federations of clusters, if the related problems (e.g. VM migration between different communication networks) can be solved.

VII. Acknowledgment

This research was funded by the french *Agence Nationale de la Recherche (ANR)* project EcoGrappe under the contract number ANR-08-SEGI-000.

References

[1] U. E. P. Agency, "Epa report to congress on server and data center energy efficiency appendices," 2007.

[2] W. Vogels, "Beyond server consolidation," *Queue*, vol. 6, no. 1, pp. 20–26, 2008.

[3] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall, "Entropy: a consolidation manager for clusters," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. New York, NY, USA: ACM, 2009, pp. 41–50.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2003, pp. 164–177.

[5] I. Habib, "Virtualization with kvm," *Linux J.*, vol. 2008, no. 166, p. 8, 2008.

[6] A. Verma, P. Ahuja, and A. Neogi, "pmapper: power and migration cost aware application placement in virtualized systems," in *Middleware '08: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc., 2008, pp. 243–264.

[7] R. Nathuji and K. Schwan, "Virtualpower: coordinated power management in virtualized enterprise systems," in *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. New York, NY, USA: ACM, 2007, pp. 265–278.

[8] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong, "Enacloud: An energy-saving application live placement approach for cloud computing environments," in *CLOUD '09: Proceedings of the 2009 IEEE International Conference on Cloud Computing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 17–24.

[9] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, and I. Scherson, "Kerrighed and data parallelism: cluster computing on single system image operating systems," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 277–286, 2004.

[10] A. Lodi, S. Martello, and D. Vigo, "Recent advances on two-dimensional bin packing problems," *Discrete Appl. Math.*, vol. 123, no. 1-3, pp. 379–396, 2002.

[11] A. K. Bhatia and S. K. Basu, "Packing bins using multi-chromosomal genetic representation and better-fit heuristic," in *Neural Information Processing, 11th International Conference, ICONIP 2004, Calcutta, India, November 22-25, 2004, Proceedings*, ser. Lecture Notes in Computer Science, N. R. Pal, N. Kasabov, R. K. Mudi, S. Pal, and S. K. Parui, Eds., vol. 3316. Springer, 2004, pp. 181–186.

[12] C.-H. Hwang and A. C.-H. Wu, "A predictive system shutdown method for energy saving of event-driven computation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 2, pp. 226–241, 2000.

[13] "Openvz." [Online]. Available: http://openvz.org/

[14] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "The xtreemfs architecture—a case for object-based file systems in grids," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 17, pp. 2049–2060, 2008.

[15] "libvirt." [Online]. Available: http://libvirt.org/

[16] "libkerrighed." [Online]. Available: http://www.kerrighed.org/

[17] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, "Stable leader election," in *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*. London, UK: Springer-Verlag, 2001, pp. 108–122.

[18] N. Bobroff, A. Kochut, and K. Beaty, "Dynamic placement of virtual machines for managing sla violations," 2007, pp. 119–128.

[19] S. Srikantaiah, A. Kansal, and F. Zhao, "Energy aware consolidation for cloud computing," in *Proceedings of HotPower '08 Workshop on Power Aware Computing and Systems*. USENIX, December 2008.

[20] M. Y. Lim, F. Rawson, T. Bletsch, and V. W. Freeh, "Padd: Power aware domain distribution," *Distributed Computing Systems, International Conference on*, vol. 0, pp. 239–247, 2009.

[21] A. Beloglazov and R. Buyya, "Energy efficient allocation of virtual machines in cloud data centers," *Cluster Computing and the Grid, IEEE International Symposium on*, vol. 0, pp. 577–578, 2010.

[22] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131.

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