A Dynamic Proportional Share Scheduler for Private Clouds

Vidya Lakshmi Rajagopalan

February 8, 2012

Abstract

To provide a resource allocation mechanism for private cloud infrastructure that can respond to the varying application resource demands as well as efficiently utilize the limited resource pool of the infrastructure is a challenge. The common models for managing cloud resources are on-demand resource provisioning and static allocation mechanisms. While the limited capacity of a private cloud makes the former model unsuitable for managing resources, the static allocation model suffers from the lack of adaptation to the time-varying application requirements and the environment's dynamicity. In this report, we propose a dynamic resource manager for private cloud platforms that is based on an economic proportional-share allocation scheme. In this model, the user communicates to the resource manager the value he is willing to spend for resources to run his applications. The resource manager allocates resources to all the applications proportional to their valuation, and inversely proportional to the resource price. The user can change the spending rate during the application runtime to take into account the changing resource requirements and also the contention state of the system. Due to its dynamic nature, this mechanism provides service differentiation to applications in contention periods, while maximizing the infrastructure resource utilization. We have implemented the resource manager in Python and integrated it with OpenNebula. Our evaluations show that our proportional share resource manager is able to perform fine grained resource allocation and provide better service to the applications paying more during contention periods.

Contents

1 Introduction

Cloud computing takes advantage of virtualization technologies to provide an on-demand resource provisioning model [6]. By using this model, users can provision (i.e., acquire and release) a dynamic number of virtual resources (i.e., virtual machine instances) anytime during their application execution based on their actual needs and for as long as they need them. Also, through the use of virtualization technologies, the applications executed by users can be isolated between each other and the resources allocated to each application can be tuned in a fine-grain manner during the application runtime. These capabilities, together with the possibility to transparently live migrate the applications between physical machines, can be used to reduce the resource fragmentation and improve the total infrastructure utilization. These benefits have encouraged the rapid adoption of the Infrastructure as a Service (IaaS) paradigm and as a follow-up a variety of cloud managers were designed to facilitate the lease and use of infrastructure resources.

Applying this on-demand provisioning model for private cloud infrastructures leads to the difficulty of managing a limited resource pool that needs to be shared between multiple competitive users. These users can execute different application types (e.g., malleable, rigid, evolving) while having different QoS requirements for them (e.g., deadline, throughput, accuracy). Thus a cloud resource manager needs to deal with the complexity of all these user demands in a scalable and efficient manner. However, cloud computing toolkits (e.g., OpenNebula [1], Eucalyptus [2]), used to manage private cloud infrastructures, rely on simple FCFS policies to allocate available resources to virtual machines. These resource managers present two main drawbacks: (i) the resource configuration for each virtual machine is specified by the user at submission time and cannot be modified during the virtual machine runtime; (ii) the resource manager does not provide any feedback to users/applications about the resource availability (i.e., how many resources the application can use) and utilization (i.e., how many resources the application used). In the first case, to run applications with time-varying per-node resource requirements the user needs to estimate the application's peak demand and overprovision the resources. In the second case, the application cannot change its resource demand according to the state of the infrastructure and its QoS goal, and they are also not given the incentives to do so.

In this report, we propose a cloud resource manager for private infrastructures that allocates fractional amounts of resources to applications dynamically during their runtimes based on a proportional-share auction that runs periodically. Auctions were seen as an appealing method to allocate resources to competitive users as they provide a generic way for the applications to express their truthful valuation for resources to the resource manager. Each application has a limited budget to spend and needs to use it wisely to request as many resources as needed. Thus the resource manager can (re-)allocate the resources to the applications that need them the most, ensuring that most valuable applications execute at the right time. We choose the proportional share allocation scheme because it is simple to implement and scalable. In this scheme, the applications receive an amount of resources proportional to what they are willing to pay for and inversely proportional with the resource price. This ensures that the resource is shared between all applications rather than a subset and it avoids starvation. We implemented the resource manager in Python and integrated it with OpenNebula. To our knowledge this is the first implementation of a dynamic proportional-share allocation mechanism for cloud infrastructures. Our evaluations show that this mechanism is able to provide differentiation between applications in contention periods.

This report is structured as follows. Section 2 describes the background and motivation for our work. The proposed proportional share strategy is described in Section 3 and Section 4 gives an overview of our implementation. We present the evaluation of this algorithm in Section 5 and the future directions and conclusions in Section 6.

2 Background

In this section, we introduce the context of our work. Then we present the drawbacks of current cloud resource allocation mechanisms and we give an overview of existing allocation schemes that could overcome them.

2.1 Context

Our system is intended for a private cloud infrastructure provider who wants to provide users with the capability to request resources for their applications dynamically during the application runtime. As compared to public clouds, private clouds have a limited resource capacity. The applications to be run can be of any type: rigid or elastic. Rigid applications are composed of a fixed number of virtual machines (VMs), determined by the user at application submission time. Elastic applications can change the number of VMs during their execution according to their computational requirements.

A central resource manager manages all the cloud resources and provides an interface to applications to provision VMs dynamically during their runtime and to partition resources in a fine-grained manner between applications. These allocations are enforced physically by a hypervisor running on each node (e.g., Xen). In this work we focus on implementing the resource allocation scheme used by the cloud resource manager. We target only homogeneous multi-core infrastructures and CPU resource allocation.

2.2 Motivation

The common way to manage cluster resources is through the use of batch schedulers. This mechanism provides users with a "job" abstraction: to run its applications, the user specifies the number of required resources, the execution duration and submits it to the scheduler queue. When enough resources become available, the scheduler allocates the requested amount of resources to the application and starts it. This model not only leads to severe resource underutilization but it also provides poor QoS guarantees to applications. As users are not allowed to change the number of resources during application runtime, they cannot react to unpredictable events (i.e., node failures) and they need to overprovision resources to meet application peak demands.

These drawbacks of static resource allocation were overcome by the dynamic provisioning model introduced by cloud computing [6]. This model allows users to "rent" virtual resources "on-demand" and scale the amount of resources according to their need. This model is attractive to use because it allows applications to optimize their resource allocation according to their need while improving the resource utilization of the infrastructure. At the same time, to use resources more efficiently, fine-grained allocation can be used [12]. With this model, the provider allocates fractional amounts of resources to applications, improving the infrastructure utilization and allowing applications that require a fixed set of processors to begin their execution sooner. However, these systems lack the capability to provide appropriate QoS guarantees to applications and to differentiate between the different priorities of user requests.

To illustrate these drawbacks, we consider the case of a user who wants to execute an elastic application, composed of a large number of independent tasks, on the cloud infrastructure without any special QoS requirement. This type is representative for data mining and semi-interactive processing applications, composed of a large number of tasks with relatively small execution times [3]. The desired behavior of the resource manager would be the following. During contention periods, the resource manager should "force" the elastic application to decrease its resource requirements letting an urgent application (e.g., a weather forecast simulation) to ask as many resources as needed and when needed. When the infrastructure is underutilized the same application should be offered more resources to have a better performance. However, common cloud resource managers (e.g., OpenNebula) are not capable of enforcing this behavior, leading to poor resource utilization and poor QoS support for the applications in need. For example, the OpenNebula scheduler relies on a first come first serve(FCFS) match-making policy to allocate resources and place the VMs on the nodes.

2.3 Resource Allocation Schemes

To solve the previously mentioned problem, the resource manager needs to know enough information about the applications running on the infrastructure (i.e., their resource demand and how much they value their QoS requirement). A generic way to achieve this is through two mechanisms: (i) sending the utility functions [9] of the applications to the resource manager; (ii) using a virtual economy to trade resources [13].

Utility functions map the current state of each application (workload, resource capacity, service level agreement) to a scalar value that quantifies the "satisfaction" of each application with regard to its performance goal. By knowing the utility functions from all applications currently running on the infrastructure, the resource manager computes an optimal allocation (e.g., maximizes the sum of all utilities or follows a max-min fairness criteria). However, this method is computationally expensive and cooperative. Users can cheat about their utilities to gain a better allocation.

Using a virtual economy to share resources between competitive users provides them with the right incentives to prioritize between their requests. Using auctions to trade resources provides a fast and efficient way to allocate resources to users that need them the most. For example, Amazon Spot instances [4] became a popular provisioning model "that allows customers to bid on unused Amazon EC2 capacity and run those instances for as long as their bid exceeds the current Spot Price. The Spot Price changes periodically based on supply and demand, and customers whose bids exceeds it gain access to the available Spot Instances." This allows the provider to sell its unused resources at a small price and improve its resource utilization. In the same time, when the demand becomes too high, only users that pay more manage to keep their desired allocations.

The simplest auction model is the proportional-share model. This type of auction allows a fine-grained allocation model at a low computational cost. With this model, the scheduler makes more efficient decissions about which applications should get the resources and which not and it also provides users with the tool to optimize their allocations according to their application importance [7]. In our work, we have used the proportional-share model for resource allocation to applications in private clouds.

Recent works used proportional share auction to allocate resources in distributed systems [11][8]. In [11], the authors present "a Dynamic Priority (DP) parallel task scheduler for Hadoop clusters which allows users to control their allocated capacity by changing the amount they are willing to spend over time." Users specify an amount they are willing to pay per time period (i.e. bid) for the execution of their tasks and the scheduler allocates to them a number of slots proportional to their bid. In [8] the proportional-share is applied per physical machine while agents bid on different machines to optimize the global allocation of the user.

Opposed to these works, we implement the proportional-share auction for allocating fractions of resources to virtual machines (VMs) in a private cloud while minimizing the number of migrations.

3 Scheduling algorithm

We designed a resource manager that uses a fine-grained dynamic scheduling model based on proportional-share auction for multi-core infrastructures. The main functionalities of the resource manager are : (i) it computes the amount of resources that each VM should get by applying a proportional-share auction on each host such that each VM receives a maximum share according to its spending rate. (ii) it computes the mapping for these vms to physical hosts so as to minimize the migrations from previous configuration.

We describe these operations in the rest of this section. For this, we first define the terms that will be used and then we introduce the proportional share auction scheme. Finally, we detail the steps taken by our algorithm.

3.1 Definitions

The proportional-share resource manager runs a resource allocation and load balancing algorithm at a predefined time period, i.e., **scheduling interval**. In our system, users are provided with an amount of virtual currency allocated by the administrator, i.e., **budget**. The budget given to each user depends on the policy adopted by the system. The users specify an amount from this budget to the resource manager, a **spending rate**, that represents how much the user wants to spend for each resource unit consumed by their applications during a scheduling interval. The spending rates are used to calculate the amount of resources that is allocated to each application. Each spending rate can be changed during the application runtime, more precisely, during each scheduling interval.

3.2 Proportional-Share Resource Allocation Scheme

We consider a cloud infrastructure composed of N nodes with a total capacity of C and a set of $1...n$ applications. The resources can be CPU capacity, memory, etc. Each application i requires NVM_i virtual machines with $VCPU_i$ cores each, for which it is willing to pay at a spending rate of SR_i. The resource allocation is done on each node according to a proportional-share mechanism. That is, a virtual machine receives a CPU share $= \frac{SR_i * C}{NVM_i * P}$ where the resource price is P $= \Sigma_{i=1}^{n} SR_i$. If each VM of an application i consumes an amount U_i of CPU, the cost charged for each application per time period is: $SR_i \cdot \Sigma_{k=1}^{NVM_i} U_i$.

This scheme avoids starvation since in each scheduling interval every application receives a portion of the cloud capacity. Since the users are charged with the amount of resources they have used, this gives them incentives to submit a spending rate that reflects the value they have for their applications. Rich users will pay more to have their application executed before a deadline, while poor users will put a low spending rate and use the resources only in low utilization periods.

3.3 A Proportional-share Algorithm for Cloud VM Allocation

The proportional share scheme calculates the proportion of resources that each VM of each application should receive, with respect to the total capacity of the cloud infrastructure. It can happen that the proportions estimated from the proportional-share scheme for the VMs of some applications with high spending rate may exceed the capacity of a node. Thus, we propose an algorithm that applies the proportional-share auction on each node of the cloud while ensuring that each VM receives the best share according to its spending rate.

We describe our algorithm next. Table 1 summarizes the data structures used for the algorithm. The algorithm is structured in three phases: (i) VM grouping; (ii) VCPU allocation; (iii) VM placement. The first phase groups the VMs in logical groups, each with a capacity of a physical node. A VM group consists of those VMs that are to be placed on the same physical node. The second phase assigns the VM VCPUs to the physical cores and computes the allocation for each of them. The last phase computes the mapping of the logical groups to the physical nodes by executing a migration plan that minimizes the number of VM migrations required and also computes the resource share of each VM according to the proportional share strategy. These phases are executed at the beginning of each scheduling interval.

Variable	Description	Fields
$NodeList$	The list of nodes	$\{nodeid, capacity\}$
$VMList$	The list of VM's with their descriptive fields	$\{vmid, spendingrate, nvcpu, share\}$
$VMGroupList$	The list of VM groups. A VM group consists of those VMs which are placed on the same physical node.	$\{vmlist, groupprice, vcpucorelist\}$ *vmlist* - List of VM's with their descriptive fields *groupprice* - Sum of spending rates of VM's in the group, i.e., the group price *vcpucorelist* - Stores the assignment of VCPU's of each VM to cores and consists of $\{coreid, coreprice, vmid, vcpu\}$ *coreid* - id of the core *coreprice* - price of the core *vmid* - id of the VM *vcpu* - vcpu id of the vcpu

Table 1: The data structures used by the proportional share algorithm

```
  Input: $VMList$,$VMGroupList$
1 Sort VMList in descending order based on the VMs spending rate ;
2 foreach group $g \in VMGroupList$ do
3     $g.groupprice = 0$ ;
4 end
5 foreach VM $v \in VMList$ do
6     Select group $g \in VMGroupList$ with the minimum price ;
7     $g.vmlist = g.vmlist \cup v$ ;
8     $g.groupprice = g.groupprice + v.spendingrate$ ;
9 end
```

Figure 1: Phase 1 of the algorithm

VM Grouping Figure 1 describes the first phase of the algorithm. In the first phase, we compute the groups of VMs that are placed on the same physical machine, according to the proportional share allocation scheme. This phase takes as an input the list of VMs with their spending rates. The list is first sorted in descending order of the VM spending rate. Then for each physical host we initialize a logical group and set its price to 0. Afterwards, the assignment of VMs to the logical groups takes place: each VM from the list is assigned to the group with the smallest price. The price of the chosen group is then incremented with the spending rate of the VM. These steps are repeated until all VMs are assigned to groups. The output of this phase is the list of VM groups.

This algorithm allocates the best possible share to each of the VM according to its spending rate, as shown in [10, page 524, theorem 20.6].

```
   Data: $VMList$,$VMGroupList$
1  foreach $group\ g \in VMGroupList$ do
2      Sort $g.vmlist$ in descending order based on the VMs spending rate ;
3  end
4  foreach $group\ g \in VMGroupList$ do
5      foreach $core\ c \in g.vcpucorelist$ do
6          $c.coreprice = 0$ ;
7      end
8  end
9  foreach $group\ g \in VMGroupList$ do
10     foreach $VM\ v \in g.vmlist$ do
11         foreach $i \in [0, v.nvcpu - 1]$ do
12             Select $c$ from $g.vcpucorelist$ with the minimum core price ;
13             $g.c.vmid = v$ $g.c.vcpu = i$
               $g.c.coreprice = g.c.coreprice + v.spendingrate$ ;
14         end
15     end
16 end
```

Figure 2: Phase 2 of algorithm

VCPU allocation The second phase of the algorithm is described in Figure 2. In this phase, the VCPUs of these VMs are mapped to the node cores. It is efficient to put the VCPUs of a VM in separate cores than in the same core since the VCPUs can be executed without any context switch. This phase begins with sorting the VMs in each group in descending order of their spending rates. Then, we set the price of each core in each group to zero. Next, for each group, in descending order of the spending rate of VMs, we execute the following step. We assign each of the VCPU of each VM to the core in that group with the smallest price. After each assignment of a VCPU to a core, the price of that core is incremented by the spending rate of the VM of the assigned VCPU. The output of this phase is the mapping of VCPU's of each VM to the cores.

VM placement Finally, the mapping of these groups of VMs to the physical nodes is performed and the resource share that has to be allocated to the VMs is computed. The simplest placement algorithm would be First Fit. However, since the algorithm is executed in each scheduling interval during which new application requests can come or existing application requests may be edited, the grouping of VMs computed in this interval can be different from that of the previous scheduling interval. As a result, some VMs need to be migrated to other physical nodes. To minimize the number of migrations, the placement of VMs to the nodes needs to consider the previous placement of VMs also. We propose a migration plan computation algorithm that minimizes the number of migrations of VMs. Figure 3 describes this phase.

```
   Data: $VMGroup$ - List of VM's to be placed together
   $node[1..n]$ - The list of physical nodes available
   $n$ - The number of physical nodes
   $chosen[1..n]$ - Represents the mapping of VMGroups to nodes
   $migration[1..n][1..n]$ - The migration matrix
 1 foreach $i \in [0, n-1]$ do
 2     if $node[i]$ didn't exist in the previous allocation round then
 3         foreach $j \in [0, n-1]$ do
 4             $migration[i][j] = 0$ ;
 5         end
 6     end
 7     else
 8         foreach $j \in [0, n-1]$ do
 9             foreach VM $v$ allocated to $node[i]$ do
10                 if $v \notin VMGroup[j]$ and exists in the current allocation
                   round then
11                     $migration[i][j] = migration[i][j] + 1$
12                 end
13             end
14         end
15     end
16 end
17 foreach $i \in [0, n-1]$ do
18     $chosen[i]$ = -1 ;
19 end
20 foreach row $k$ in migration do
21     $p = migration[k][j]$, which is the smallest in the row $migration[k]$ ;
22     if $\exists m$ such that $chosen[m] = j$ then
23         $p_1 = migration[k][j_1]$ such that $p_1$ is the next highest element
           in row $k$ to $p$. ;
24         $q = migration[m][j]$. ;
25         $q_1 = migration[m][j_2]$ such that $q_1$ is the next highest element
           in row $m$ after $q$ ;
26         if $p + q_1 < q + p_1$ then
27             $chosen[k] = j$  Go to step 22 with $k = m$ and $j$=$j_2$
28         end
29         else
30             Go to step 22 with $k = k$ and $j$=$j_1$
31         end
32
33     end
34     else
35         $chosen[k] = j$
36     end
37 end
38 foreach $i \in [0, n-1]$ do
39     foreach VM $v$ allocated to $node[i]$ do
40         g = the group to which v belongs ;
41         $v.share = (v.spendingrate/g.groupprice) * node[i].capacity$
42     end
43 end
```

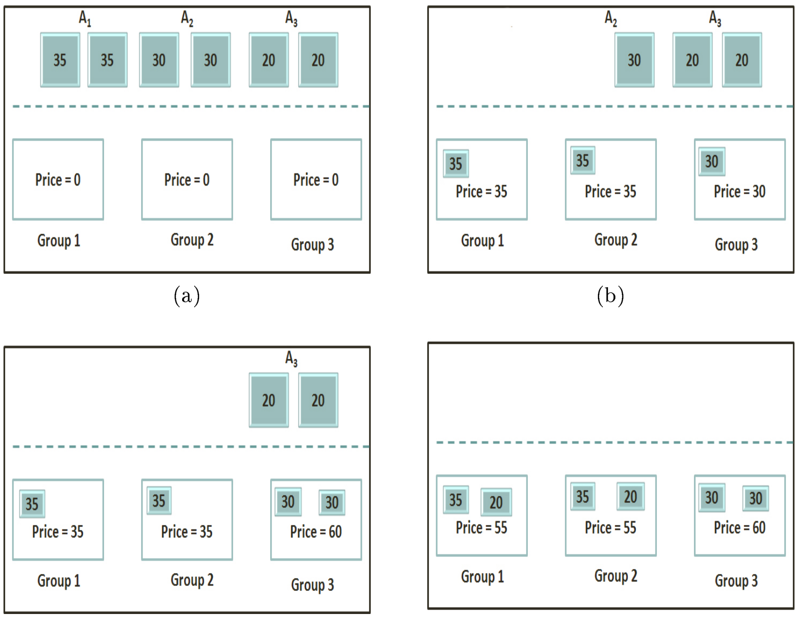
Figure 3: Phase 3 of algorithm

Lines 1 to 15 of the algorithm compute the migration matrix, $migration[][]$. The migration matrix keeps track of the number VM migrations required for each node, when a certain VMGroup is placed on it. Each row of the matrix corresponds to each node, and the columns correspond to each $VMGroup$ For example, the element $migration[i][j]$ gives the number of VMs that has to be migrated from the i'th node when the VM group j is allocated to it. Lines 2 to 15 are performed for each $node[i]$. If $node[i]$ has been added in the current scheduling interval, we set all elements corresponding to the row $migration[i]$ to 0 since there are no VMs that needs to be migrated from $node[i]$. Otherwise, we check if each of the VM v that was allocated to $node[i]$ in the previous round, doesn't belong to $VMGroup[j]$ and exists in the current allocation round. If so, it means that placing $VMGroup[j]$ on $node[i]$ would lead to the removal of VM v and thus to a migration. Hence, we increment the element $migration[i][j]$.

Lines 20 to 36 describe the mapping of the VM groups to the nodes. This mapping is stored in the array *chosen*. For each row in the migration matrix, the column which contains the minimum element in that row gives the VMGroup that when placed on that node, leads to a minimum number of migrations. If the VM group is already assigned to another node, then the element that is the next higher to the minimum is found and the steps to find the allocation with minimum migrations is computed as shown in figures 22-33. We consider that at this step the VM group that was the best assignment for node i was already assigned to node m. Basically this translates in the fact that for row i, the j^{th} column element is the minimum, but the j^{th} column is already assigned to row m. In this case we try to see if keeping the assignment of VM group j to node i and assigning another VM group to node m is better (Lines 22-33). For this, we find the elements $migration[m][j]$ and $migration[m][j_1]$ such that $migration[m][j_1]$ is the next highest element to $migration[m][j]$ in row m. Similarly, we find elements $migration[i][j]$ and $migration[i][j_2]$ such that $migration[i][j_2]$ is the next highest element to $migration[i][j]$ in row i. If the total number of migrations from this exchange is less than assigning another VM group to the node i (the sum of migration[i][j] and migration[m][j1] is less than the sum of migration[i][j2] and migration[m][j]), then the VM group j is assigned to node i (Line 27) and the steps are re-taken to find a new mapping for node m. Otherwise, the same steps are re-taken to check the new mapping of VM group j_2 to node i. At the end of this phase, the CPU shares are computed for each VM (lines 38-41).

Example We illustrate the algorithm through the following use case. We consider an infrastructure with 3 physical nodes with 4 cores each. Thus the total infrastructure capacity is C=1200. 3 applications are submitted to the infrastructure. The application requests are of the form (spendingrate, number of VMs, number of VCPUs) and are the following: $A_1 - (70, 2, 3)$; $A_2 - (60, 2, 3)$; $A_3 - (40, 2, 3)$. The total resource price is: $Price_{system} = 70 + 60 + 40 = 170$

Figure 4a represents the initial scenario where each group of nodes is initialized with a 0 price. Figure 4b and Figure 4c show the execution of phase 1 where the VM with highest spending rate is assigned to the group with the least price. To start with, in our example, the VM with the highest spending rate(35), is assigned to the group with the least price(0). After this, the spending rate of the assigned VM(35) is added to the price



(a) (b) (c) (d)

Figure 4: Phase 1 - Grouping of VMs

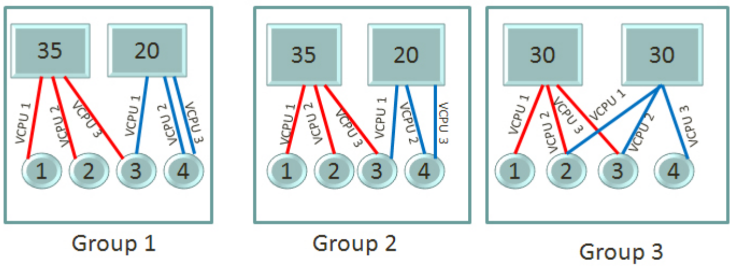


Figure 5: Phase 2 - VCPU to core assignment

of VM group 1. If there are multiple VMs with the same highest price, the first VM is chosen. Similarly, if there are multiple groups with the smallest price, the first group is chosen. Then, the algorithm proceeds to assign the remaining VMs to the VM groups.

Figure 5 shows the assignment of VCPUs to the physical cores. The circles represent the cores of each physical node and the lines represent the VCPU of a VM which is assigned to a particular core. The first group is chosen and the price of all the cores are set to zero. The VM with the highest spending rate is chosen and each of its VCPU is assigned to the core with the least price. This is done until all VCPUs are assigned to cores. Similarly, this is done for groups 2 and 3.

4 Implementation

In this section, we describe the details of the implementation. We first give an overview of the system and then describe the resource manager configuration and its main execution steps.

4.1 System Overview

Figure 6 shows the architecture of our system. Users submit the requests to a central resource manager(the proportional-share scheduler) that resides on the frontend of the cloud. The resource manager provisions the VMs from a virtual infrastructure manager(OpenNebula). OpenNebula is an opensource toolkit for cloud infrastructures. The main functionalities provided by OpenNebula are user and physical host management, VM and virtual network management. Users can provision VMs in the cloud by submitting to OpenNebula a template file, which describes the VM requirements like memory and CPU. OpenNebula has a resource manager daemon which selects the physical nodes to deploy the VMs and allocates resources like CPU capacity, memory to the VMs. The resource manager in OpenNebula is an independent daemon which can be replaced by third party schedulers. We implemented our resource manager and integrated it with OpenNebula using the provided **XML-RPC interface**.

We have implemented the Proportional share resource manager in **Python**. For the interaction with OpenNebula, we used a part of the Haizea code [5]. The XML-RPC calls, the datastructures to store the parameters obtained from the calls are inspired from Haizea code.
The enforcement of physical allocations was done through the Xen hypervisor that resided on each OpenNebula compute node. We chose Xen because it provides fine grained allocation control of CPU to the VMs and can also pin the VCPUs of each VM to cores. To store the application requests, we use a MySQL database on the OpenNebula frontend.

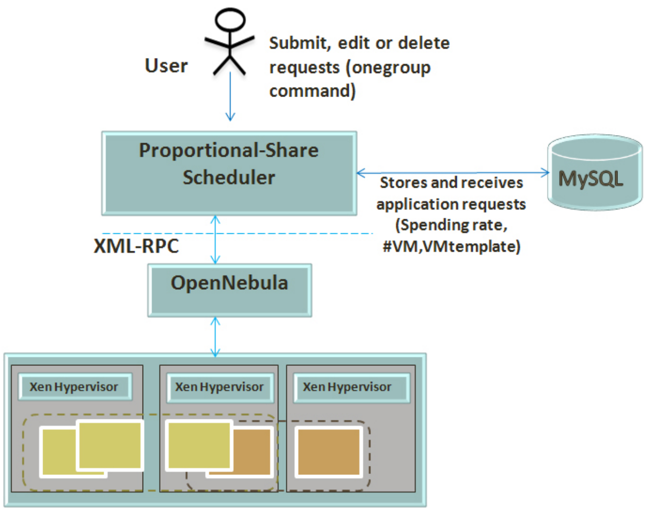

User
Submit, edit or delete requests (onegroup command)
Proportional-Share Scheduler
Stores and receives application requests (Spending rate, #VM,VMtemplate)
MySQL
XML-RPC
OpenNebula
Xen Hypervisor
Xen Hypervisor
Xen Hypervisor

Figure 6: System Architecture

4.2 Resource Manager Configuration

The resource manager has a set of parameters that can be modified by the system administrator. These parameters are included in a configuration file named defaults.py. Table 2 describes all the parameters that can be tuned. To start the resource manager, we execute the command,
scheduler start
and to stop the resource manager, we execute the command,
scheduler stop

Parameter	**Description**
LOGFILE	The file to which the logs of the resource manager is to be written
ON_RPC_PORT	The XML-RPC server port of OpenNebula
ON_RPC_HOST	The host on which the XML-RPC server is running
ON_XMLRPC_URI	The URI used to connect to the OpenNebula XML-RPC server
MYSQL_SERVER	The host where MYSQL server is running
MYSQL_USER	The username to login to the MYSQL server
MYSQL_PASSWD	The password of MYSQL_USER to login to the MYSQL server
WAITTIME	The scheduling interval of the resource manager in seconds
DEPLOYTIME	The time(in seconds) that the resource manager needs to sleep to ensure that the VMs have been deployed and starts running at a node. This time depends on the time it takes for OpenNebula to deploy VMs in nodes.

Table 2: The configuration parameters of the resource manager.

4.3 Resource Manager-System Interaction

The resource manager once started executes in rounds. The scheduling interval can be set using the WAITTIME configuration parameter. Once started, the resource manager daemon executes the following steps :

1. It creates the required tables in the database *scheduler*, if they don't exist. The application request data is stored in two tables, *groups* and *vms*. The schema of the tables are shown in Table 3 and Table 4.
2. It connects to the XML RPC server of OpenNebula.
3. It retreives from OpenNebula the current host configuration. This involves invoking the XML-RPC call *one.hostpool.info*.
4. It retreives from the database *scheduler* the list of submitted application requests and identifies if they are already submitted VM requests or edited requests(using the *changeparam* column in *vms* table) or if they are newly submitted VM requests. Then, it retrieves the VM details from OpenNebula using XML-RPC call *one.vm.info*.
5. If there is at least one physical node in the cloud and there are new or edited VM requests, it executes steps 6 to 7.

6. It executes the algorithm described in pseudocode in Section 3. First, phase 1 of the algorithm is executed and the VMs that are to be placed together in the same node are found. Then, it executes phase 2 of the algorithm and the mapping of the VCPUs of the VMs to the cores of the nodes is performed. Finally, it executes the final phase and the group of VMs are mapped to the physical nodes. In the first round of resource manager execution, the allocation is done in round robin way. Starting with the second round, the re-mapping of VMs to physical nodes is performed. Also, the resource share that each VM receives is computed at the end of this phase.
7. The VM's are deployed on the nodes. There are three different cases in this scenario. i) New VM requests ii) Allocation change for existing VMs iii) Allocation change for existing VMs that have to be migrated. In case of new VM requests, they are deployed in the nodes. This is done executing the *one.vm.deploy* XML-RPC call. The resource manager waits *DEPLOYTIME* seconds, which ensures that the deployment of the VMs on nodes is done and the VMs are running. In case of VMs that are already running, the resource manager handles the following cases: (i) if the VMs need to be migrated to a new node, the resource manager invokes the migrate operation of OpenNebula by executing the *one.vm.migrate* XML-RPC call, sets the CPU share for each VM and pins the VCPUs to the cores. (ii) if only their VM resource allocation changes, the resource manager sets the resources accordingly. In all the three cases mentioned above, the resource manager sets the cpu shares and pins the VCPUs to the cores by communicating with the Xen VMM of each host.
8. The resource manager sleeps for *WAITTIME* seconds.
9. Step 3 is repeated.

Column	**Type**
groupid	int (primary key)
vmtemplate	varchar(1000)
spendingrate	double
nvm	int

Table 3: Schema of table *vms*

Column	**Type**
vmid	int (primary key)
groupid	int (foreign key)
changeparam	int

Table 4: Schema of table *groups*

5 Evaluation

In this section, we describe the experiments we ran to study the allocation dynamics of our proportional share resource manager. We performed an experiment to measure the performance of the application with different spending rates. In the following subsections, we describe the setup and results we obtained.

5.1 Evaluation setup

The experiments were run on the Grid'5000 testbed. We used 10 nodes with 8 cores as the compute nodes of the cloud. The nodes were configured with Debian images with support for the Xen VMM.

We consider the case when one or more users submits a benchmark application with different spending rates. We selected the NAS Benchmark Application which generates independent Gaussian random variates using the Marsaglia Polar Method. This is an embarassangly parallel application.

We split the user requests to four different classes which are described in the Table 5 20 such requests were submitted in two rounds. The

Class of Application	**(Spending rate, #VM, #VCPU)**
Class 1	(30,1,8)
Class 2	(60,1,8)
Class 3	(90,1,8)
Class 4	(120,1,8)

Table 5: The relation between the request parameters and the application class

applications start running as soon as the VM's are booted.

5.2 Results

Figure 7 shows the distribution of execution times for each request class. The results show that the applications with higher spending rate com-

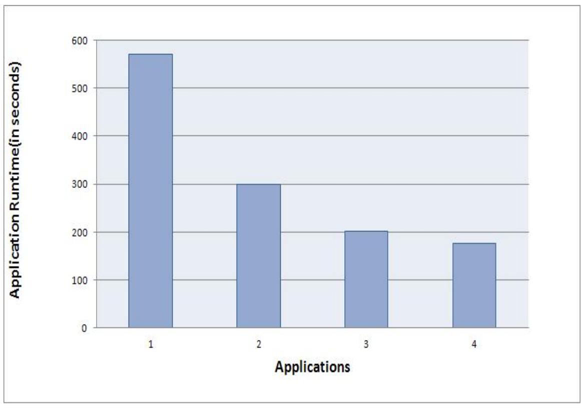


Figure 7: Execution time for different applications

plete faster than the others. Thus, the proportional share algorithm is fair, since the share of CPU allocated to each application is relatively according to its spending rate.

We also evaluated the improvement brought by our VM placing algorithm. We found that with our migration algorithm, 4 VM's were migrated while with the "naive" round robin placement scheme 8 VM's had to be migrated. Thus, our node allocation algorithm minimizes the number of migrations of VMs compared to classic allocation strategies like round robin.

6 Conclusion

Cloud computing infrastructure providers became popular due to the provisioning model they offer to users. This provisioning model is attractive, as users can lease and release virtual machines on demand, for as much and as long their application needs them. However, current cloud resource managers lack the capability to distinguish between the priorities of user requests. Users needing resources more than others cannot make the resource manager aware of their importance. Also, their inability to provide feedback about the resource availability and resource utilization makes it difficult for the applications to adapt their resource requirements according to the state of the infrastructure and their QoS goal.

In this report we proposed a dynamic resource manager for private cloud infrastructures that allocates resources to virtual machines dynamically during their runtimes and provides feedback to applications regarding the infrastructure's resource availability. Our resource manager relies on a proportional-share auction model to partition resources between applications. To our knowledge this is the first implementation of proportional-share allocation for VMs in a private cloud. The pricing and bidding model used by our resource manager gives a fair, easy to use and scalable solution for resource allocation. Moreover, we take advantage of the multi-core nodes to provide a more fine-grained allocation of resources to virtual machines. Thereby, our resource manager is able to provide a better utilization of the available resources in a private cloud. We have implemented a prototype of the resource manager and integrated it with OpenNebula. We have performed real experiments on Grid'5000. Our experiments have shown that the proposed resource manager is able to provide service differentiation to applications based on their priorities. Our work opens up new perspectives for designing policies to meet application performance goals on such an infrastructure.

As future work, we plan to extend the current mechanism to manage heterogeneous infrastructures and multiple resource types. For a more scalable solution, we will investigate methods of decentralizing the current implementation. Finally, we plan to evaluate the scalability of the mechanism on a larger testbed.

Acknowledgements

This report is the result of an internship done under the guidance of Stefania Costache, Nikos Parlavantzas and Christine Morin in the Myriads team at INRIA, Rennes Bretagne-Atlantique, France. The duration of the internship was from June 2011 to August 2011.

References

[1] http://opennebula.org/.

[2] http://www.eucalyptus.com/.

[3] http://en.wikipedia.org/wiki/MapReduce/.

[4] http://aws.amazon.com/ec2/spot-instances/.

[5] http://haizea.cs.uchicago.edu/.

[6] AmazonEC2. http://aws.amazon.com/ec2/.

[7] K. Lai. Markets are dead, long live markets. *ACM SIGecom Exchanges*, 5(4):1–10, 2005.

[8] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B.A. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[9] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. SLA-aware virtual resource management for cloud infrastructures. In *9th IEEE International Conference on Computer and Information Technology (CIT'09) 9th IEEE International Conference on Computer and Information Technology (CIT'09)*, pages 1–8, 2009.

[10] Eva Tardos Vijay V. Vazirani Noam Nisan, Tim Roughgarden. *Algorithmic Game Theory.*

[11] Thomas Sandholm and Kevin Lai. Dynamic proportional share scheduling in hadoop. In *15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.

[12] M. Stillwell, D. Schanzenbach, F. Vivien, and H. Casanova. Resource Allocation Using Virtual Clusters. In *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2009.

[13] Chee Shin Yeo and Rajkumar Buyya. A taxonomy of market-based resource management systems for utility-driven cluster computing. *Softw. Pract. Exper.*, 36:1381–1419, 2006.

Appendix A - User Manual

In this section we describe the interface that is provided to the users to request resources for their applications and the steps required to install the resource manager on the OpenNebula frontend. The structure of an application request consist of the VM template, spending rate and the number of VM's. The user can create,edit,list and delete the requests. Correspondingly, VMs are created or deleted through the XML-RPC interface of the OpenNebula.

The following are the steps to be followed to use the proportional share resource manager, assuming that OpenNebula is already installed in the frontend machine and the OpenNebula scheduler is not running.

1. Place the scheduler folder in a directory of your choice and set the PATH variable to that scheduler directory. The *scheduler* daemon(which includes onegroup command also) should be installed in the frontend of the OpenNebula machine since the daemon uses OpenNebula user permissions to SSH to the physical nodes in the cloud.
2. Make sure that the OpenNebula ONE_AUTH env variable is set.
3. Install MySQL server in any machine and create database *scheduler* in it. Make sure that the *resource manager* daemon is be able to connect to the MySQL server.
4. Set the parameters in *defaults.py* with appropriate values.
5. Use the *onegroup* command to create,edit,list or delete application request. The usage of the *onegroup* command is :

onegroup [OPTIONS] [create/edit/delete/list]
-t –template
-s –srate
-n –nvm

Table 6 describes the usage of *onegroup* command.

6. To start the resource manager execute
   scheduler start
7. To stop the resource manager execute
   scheduler stop
8. View the logs of the resource manager in LOGFILE defined in defaults.py

Command	**Description of parameters**	**Output**	**XML-RPC call**
onegroup create -t *template* -s *srate* -n *nvm*	*template* - VM template *srate* - spending rate *nvm* - Number of VMs	Creates a request with the specified parameters. It assigns an id to the request stored as *groupid*	one.vm.allocate
onegroup list	-	Lists all the request with values - groupid, spending rate, #VM, vmid(ID of VMs created for that request)	-
onegroup edit *groupid* -s *srate* -n *nvm*	*groupid* - id of the request *srate* - spending rate *nvm* - Number of VMs	Edits the parameters associated with the request with id *groupid*	one.vm.action with parameter 'finalize' one.vm.allocate
onegroup delete *groupid*	*groupid* - id of the request	Deletes the request with the id *groupid*	one.vm.action with parameter 'finalize'

Table 6: Different usages of *onegroup* command