Green-DAM: a Power-Aware Self-Organizing Approach for Cloud Infrastructure Management

Daniela Loreti[1] and Anna Ciampolini[1]
[1]*Department of Computer Science and Engineering, University of Bologna, Bologna, Italy*
{*daniela.loreti, anna.ciampolini*}*@unibo.it*

Keywords: Green Computing, Distributed Cloud Infrastructure Management

Abstract: As the number of cloud users grows up, the rate of datacenter's infrastructure complexity goes higher and higher. For this reason a lot of efforts are now concentrated on providing the cloud paradigm with an autonomic behavior, so that it can take decisions about virtual machine allocation and migration across the datacenter's nodes without human intervention. While the major part of these solutions is intrinsically centralized and suffer of scalability and reliability problems, we want to investigate the possibility to provide the cloud with a decentralized self-organizing behavior. We present a novel communication protocol to exchange status informations between hosts and reach a decentralized reallocation plan. To test the protocol behavior, we developed a simulator able to compare the performance of a given policy applied through a decentralized approach with the correspondent centralized solution.

1 INTRODUCTION

The Cloud Computing paradigm experienced a significant diffusion during last few years thanks to its capability of relieving companies of the burden of managing their IT infrastructure. At the same time the demand for efficient yet scalable cloud architectures makes the Green Computing area stronger, driven by the pressing need for greater computational power and for restraining economical and environmental expenditures.

The challenge of efficiently managing a collection of physical servers avoiding bottlenecks and power waste, is not solved at all by Cloud Computing paradigm, but only partially moved from customers's IT infrastructure to provider's big data centers. Since cloud resources are often managed and offered to customers through a collection of virtual machines, a lot of efforts concerning Cloud Computing paradigm are concentrating on finding the best virtual machine allocation to obtain efficiency without compromising performances.

Giving that an idle machine is demonstrated to consume around 70% of it peak power (Fan et al., 2007), packaging the virtual machines into the lowest possible number of servers and switching off the idle ones, can lead to a higher rate of power efficiency, but can also cause performance degradation in customers's experience and Service Level Agreements (SLAs) violations.

On the other hand, allocating virtual machines in a way that the total cloud load is balanced across different nodes, will result in a higher service reliability and less SLAs violations, but forces the cloud provider to maintain all the physical machines switched on and, consequently, cause unbearable power consumption and excessive costs.

In addition, we must take into account that such a system is continuously evolving: demand of application services, computational load and storage can quickly increase or decrease during execution.

Due to these contrasting targets the virtual machine management in a Cloud Computing datacenter is intrinsically very complex and can be hardly solved by a human system administrator, making more and more desirable to provide the infrastructure with the ability to operate and react to dynamic changes without human intervention.

The major part of the efforts in this field relays on centralized solutions, in which a particular node of the cloud is in charge of collecting informations on the whole set of physical hosts, taking decisions about virtual machine allocation or migration, and operating to apply these changes on the infrastructure (Jung, 2010; Lim et al., 2009). The advantages of these centralized solution are well known: a single node with complete knowledge of the infrastructure can take better decisions and apply it through a restricted number of migrations and communications. However scalability and reliability problems of centralized solutions are known as well. Furthermore as the number of physical servers and virtual machines grows, solving the allocation problem and finding the optimal solution can be time expensive, so typically some other approximation algorithm is used to reach a sub-optimal solution in a fair computation time (Beloglazov and Buyya, 2012).

In this work we investigate the possibility of bringing allocation and migration decisions to a decentralized level allowing the datacenter's physical nodes to exchange informations about their current virtual machine allocation and self-organize to reach a common virtual machine reallocation plan.

Even if we know this kind of systems probably cannot lead to a complete distributed solution due to intrinsic centralized features of cloud infrastructure, we believe that a higher rate of distribution in virtual machine allocation can bring a reduction in system administration complexity and lead to a more scalable and reliable solution.

We propose a novel decentralized way to apply a virtual machine migration policy to the cloud: we imagine the datacenter as partitioned into a collection of overlapping neighborhoods, in each of which a local reallocation strategy is applied. Taking advantage from the overlapping, the virtual machine redistribution plan propagates on the global cloud.

We analyze the effects of this approach by comparing them with the centralized application of the same policy. In particular we focus on the definition of the Distributed Autonomic Migration (DAM) protocol, used by cloud's physical hosts to communicate and get a common decision as regards the reallocation of virtual machines, according to a predefined global goal (e.g. power-saving, load balancing, etc.). In particular, in this work we adopt Green-DAM: a policy whose goal is to save power, while maintaining the load balanced across the physical server network.

We tested our approach with power saving objectives, implementing Green-DAM by means of DAM-Sim: a software to simulate the behavior of different policies applied in a traditional centralized way or through DAM protocol on a decentralized infrastructure.

The article is organized as follows: in section 2 we present the architecture of our solution, in section 3 we show the experimental results obtained by means of the DAM-Sim simulator, section 4 shows the state-of-the-art of Cloud Computing infrastructure management and section 5 illustrates our conclusions and future works.

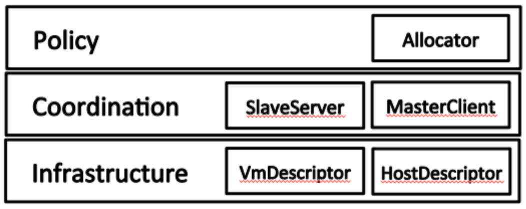


Figure 1: The three tier architecture of the Sim-DAM simulator.

2 ARCHITECTURAL FRAMEWORK

We present a distributed solution for Cloud Computing infrastructure management, with a special focus on virtual machine allocation and migration. Our approach is built on a model, which keeps separated the server coordination protocol from the specific migration policy implemented. In this way different goals can be followed by only changing the adopted policy while the communication model remains the same.

As shown in figure 1, the framework is composed of three main layers:

- the infrastructure layer, specifying a software representation of the cloud's entities (e.g. hosts, virtual machines, etc...);
- the coordination layer implementing DAM protocol, which defines how physical hosts can exchange their status and coordinate their work;
- the policy layer, containing the rules that every node must follow to decide where to possibly move virtual machines.

We describe each layer in the following sessions.

2.1 Infrastructure Layer

The infrastructure layer defines which informations must be collected about each host's status. To this purpose two basic structures are maintained:

- *HostDescriptor*,
- *VmDescriptor*.

As shown in figure 2 the *VmDescriptor* contains the maximum MIPS value a virtual machine can request (T_{vm}), the percentage of these MIPS currently in use (m_{vm}) and the id of the node in which the virtual machine is currently hosted (*currentHostId*). It also includes a *futureHostId* value representing the id of the node in which the virtual machine can be migrated at the end of the protocol execution.

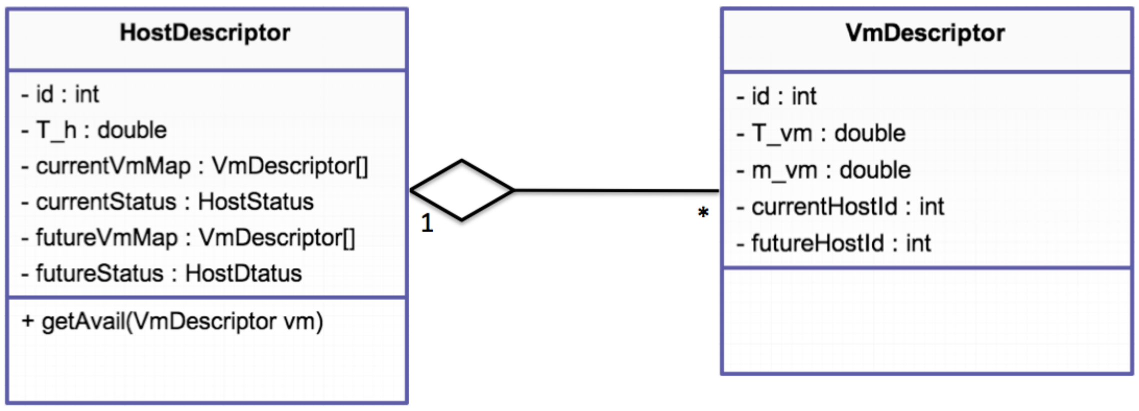

HostDescriptor
- id : int
- T_h : double
- currentVmMap : VmDescriptor[]
- currentStatus : HostStatus
- futureVmMap : VmDescriptor[]
- futureStatus : HostDtatus
+ getAvail(VmDescriptor vm)
1
*
VmDescriptor
- id : int
- T_vm : double
- m_vm : double
- currentHostId : int
- futureHostId : int

Figure 2: UML diagram of the host and virtual machine descriptors.

Each *HostDescriptor* includes the amount of total CPU MIPS (see section 2.1.1) the node can execute (T_{vm}) and two collections of *VmDescriptors*:

- *currentVmMap* indicates which virtual machines are currently allocated on the host; while
- *futureVmMap* collects the virtual machines that will be allocated on the host when a common decision is reached by the distributed system.

The *futureVmMap* is a sort of temporary copy of the status, exchanged through hosts according to the DAM protocol. We will discuss the need for this copy of the future status in section 2.2.

The *HostDescriptor* also includes a *currentHostStatus* variable representing its current state. The possible values of *currentHostStatus* are listed below:

1. OFF: no operation running, because no virtual machine is hosted. [1]
2. UNDER: the host is switched on but executes too few operations, so a probable energy waste is detected. To simplify our model we consider host in UNDER status when the amount of executed MIPS is over a threshold we call THRES_DOWN, but other more complex conditions can be easily implemented into the architecture (e.g: to detect an underloaded condition of the host in a more elaborate and predictive way).
3. OVER: the host is executing too much MIPS risking to run into SLAs violations. Similarly to UNDER, the OVER status is detected when the percentage of used MIPS goes over a threshold (THRESH_UP).
4. OK: the host is switched on and it is supposed to work without power wastes or SLA violation.

[1]Despite the name of this status, we imagine a host will never be completely switched off in a real environment, otherwise, when it turns out it is needed again, it will takes a lot of time to switch it back on.

Similarly to the collection of *VmDescriptors*, we have two *HostStatus* stored into the *HostDescriptor* to map the current status (*currentHostStatus*) and a copy (*futureHostStatus*) representing the future status assigned to the host by the distributed management computation.

2.1.1 The CPU model

The amount U_h of CPU MIPS used by the host h is calculated as follows:

$$U_h = \sum_{vm \in currentVmMap_h} m_{vm} \frac{T_{vm}}{100} \qquad (1)$$

where $currentVmMap_h$ is the set of virtual machines currently allocated on host h, T_{vm} is the total CPU MIPS vm can request and m_{vm} is the percentage of this total that vm is currently using.

Indeed we consider a simplified model in which the total MIPS executed by the node can be seen as the sum of MIPS used by each hosted virtual machine.

2.2 Coordination Layer

The coordination layer implements the DAM protocol, which defines the sequence of messages that hosts must exchange in order to get a common migration decision and realize the defined policy.

The protocol is based on the assumption that the datacenter is divided into a predefined fixed collection of overlapping subsets of hosts: we call every subset a *neighborhood*.

We assume that no change involves the neighborhood network during the protocol execution, so that in every moment each host can only communicate with

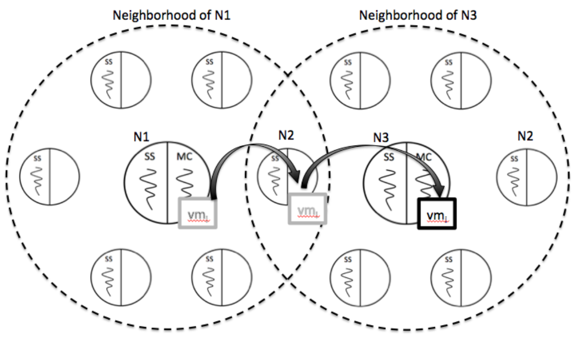


Figure 3: Schema of two overlapping neighborhoods.

the same neighborhood of nodes. As shown in figure 3, we assume that each physical host executes a demon process called *SlaveServer*, which has a copy of the node's status stored into an *HostDescriptor* and can send it to every other node asking for that.

According to the chosen policy each node can monitor its computational load and the amount of resources used by each virtual machine it hosts and decide either it is in a critical condition or not. In our implementation a node can for example detect to be in UNDER status when the sum of the MIPS executed by the virtual machines hosted is not enough to exceed the THRESH_DOWN threshold. If this happens, the node starts another process, the *MasterClient*, to actually make a protocol interaction begin. We call *rising condition* the one that turns on a node's *MasterClient*.

Since there is a certain rate of overlapping between neighborhoods, the information about the migration of a virtual machine can be spread through different neighborhoods.

To better explain the coordination between nodes, figure 3 shows an example of two overlapping neighborhoods. Each node has a *SlaveServer* always running to answer questions from other node's *MasterClient*, and optionally can also have a *MasterClient* process started to handle a critical situation by starting the DAM protocol. A virtual machine *vm* allocated to an underloaded node N1 can be moved out of it on N2 and, as a consequence of the execution of the protocol in the adjacent neighborhood of N3, it can be moved again from N2 to N3. It is worth to notice that node N2, as each node of the datacenter, would have its own fixed neighborhood, but it starts to interact with it by means of a *MasterClient* only if a *rising condition* is observed.

Note that N1's *MasterClient* must have N2 in its neighborhood to interact with it, but the *SlaveServer* of N2 can answer to requests by any *MasterClient* and, if a critical situation is detected and the *MasterClient* of N2 is started, its neighborhood does not necessarily include N1.

If as a consequence of the application of the policy on a neighborhood, all the virtual machines on a host can be migrated away; then, the node can be put in sleep mode to save power.

As regards this environment we must remark that the migration policy (and the *rising condition*) should be chosen conveniently to prevent never-ending cycles in the migration process. We will discuss the characteristics the policy should have in section 2.3.1

2.2.1 Coordination details

The protocol must ensure that the neighbors's states the MasterClient obtain is consistent from the beginning to the end of the interaction. Therefore the protocol is made of two phases we firstly illustrate from the *MasterClient* point of view:

Listing 1: MasterClient code

```
//Input: int MAXround, Allocator
    allocator
  int round=0;
  NodeList neighList;
  HostDescriptor[] pastHDCollection;
  HostDescriptor[] HDCollection;
  while(true){
//Phase 1:
    foreach ss in neigh{
      send(ss,"lock");
      HostDescriptor hd=receive(ss);
      HDCollection.add(hd);
    }
//Phase 2:
    if(HDCollection==
        pastHDCollection){
      round++;
    }else {
      round=0;
      pastHDCollection =
          HDCollection.clone();
    }
    if (round<MAXround){
//Phase 2A:
        allocator.optimize(
            HDCollection); //policy
        foreach ss in neigh{
          send(ss,HDcollection.get(
              ss)); //and unlock
        }
    }else{
//Phase 2B:
      foreach ss in neigh{
        send(ss,"update-current-
            status"); //and unlock
      }
          break;
    }
  } //while
```

In phase 1 the *MasterClient* acquires the lock from every *SlaveServer* neighbor sending a *"lock"* message to it (line 9 in listing 1). To prevent deadlock each host of the datacenter is marked with a numeric identifier such that the *MasterClient* can lock the *SlaveServer*s (included the one on the machine he runs on) by ID order. After each lock message sent, the *MasterClient* waits for the *SlaveServer* to send back a message containing its *HostDescriptor*. The *MasterClient* stores the *HostDescriptors* from its neighborhood into a collection (lines 10-11)

If a *SlaveServer* neighbor is held in another interaction does not send its *HostDescriptor* until the precedent protocol round is end and the *MasterClient* is suspended waiting for the answer. This ensures that the *SlaveServer*'s *HostDescriptor* is never changed concurrently by two different *MasterClient*s.

In phase 2 the *MasterClient* compares all the received *HostDescriptor*s in *HDcollection* with the previous copy in *pastHDcollection*. If they are all unchanged, this means that probably no other *MasterClient* is performing a round of the protocol in the meanwhile, so a *round* counter is incremented otherwise the received *HDcollection* is stored in the *pastHDcollection* copy (line 18).

When *MasterClient* has collected informations about all the *SlaveServer* neighbors, can start to compute a virtual machine reallocation according to the defined policy (Phase 2A). In order to do so the *MasterClient* calls the *optimize* operation (line 22) passing to the policy allocator object the collection of*HostDescriptors* of his neighbors. This mechanism ensures a separation between the DAM protocol and the policy so that the same communication model can be used either to consolidate virtual machines on servers and save power, or to balance the computation load and preserve SLAs.

In phase 2A the *MasterClient* also sends back to each *SlaveServer* neighbor the *HostDescriptor* modified according to the policy (line 24), always respecting the nodeID-order. The state is accepted passively, without contradictory. Migration decisions performed by the *optimize* operation, only changes the future copy of the host's state and the collection of future virtual machine's descriptors (*futureHostStatus* and *futureVmMap* in figure 2). To ensure that, the *SlaveServer* overwrite his HostDescriptor without performing changes into the real configuration. No host switch on/off or virtual machine migration is performed in this phase.

If the number of round with unchanged neighbors configuration exceeds a defined maximum, phase 2B is executed: the *MasterClient* sends to all *SlaveServer* a *update-current-status* request to notify the *SlaveServer* that the informations on HostDescriptor should be applied to the real system state. The *SlaveServer* again execute it passively. In our GreenDAM simulator this is implemented by coping the *HostDescriptor*'s *futureStatus* into the *currentStatus* and the collection of virtual machine descriptors *futureVmMap* into the *currentVmMap*. In this way we simulate migrations physically performed.

Phase 2A and 2B alternatives comes from the need for reducing the number of migration fiscally performed. Looking at example in figure 3, if hosts only exchanged and updated the current collection of virtual machines, every *MasterClient* can only order a real migration at each round, so that vm_i on N1 would be migrated on N2 at first, and later on N3. Using a *futureVmMap* and a future state (initially copied from the real ones) and performing all the reallocation policy on this abstract copy, real migration are executed only when the N3's *MasterClient* exceeds *MAXround* and vm_i can directly go from N1 to N3.

The corresponding code executed by a *SlaveServer* is shown below.

Listing 2: SlaveServer code

```
 1 //Input: RisingCondition rc,
       HostDescriptor hd
 2   if (rc.check())
 3     startMasterClient();
 4   while(true){
 5 //Phase 1:
 6     receive(masterClient, "lock");
 7     lockMyState();
 8     send(mc, hd);
 9 //Phase 2:
10     receive(mc, element);
11     if (element!="update-current-
           status"){
12 //Phase 2A:
13       hd = (HostDescriptor) element;
14     }else {
15 //Phase 2B:
16       updateCurrStatus(element);
17     }
18     unlockMyState();
19     if (rc.check())
20       startMasterClient();
21   } //while
```

The *SlaveServer* code mirrors the *MasterClient* one, but starts with a *rising condition* check because the first operation the *SlaveServer* must do is to check its state: if detects a critical situation starts a *MasterClient* itself to handle it (line 3).

In phase 1 the *SlaveServer* suspend itself on receiving a lock by any *MasterClient* (line 6). Note that differently from the *MasterClient*, who can talk only with its neighbors, the *SlaveServer* can answer to every other node of the cloud. Indeed the neighborhood's structure is held only by the *MasterClient*s.

When the *SlaveServer* receives a *lock* request, locks itself (line 7) so that other *MasterClient*s cannot proceed until this *SlaveServer* is unlocked and then send its *HostDescriptor*.

In phase 2 the *SlaveServer* waits for a message back and control either it is an update of the *HostDescriptor* just sent (phase 2A) or a request for a current state update (phase 2B). In case 2An the *HostDescriptor* is overwritten without contradictory. Remember that, as a consequence of the *MasterClient*'s behavior, this new *HostDescriptor* only changed the *futureStatus* and *futureVmMap*, while in phase 2B this values ace used to perform real migrations.

The *VmDescriptor* contains both the *currentHostId* and the *futureHostId* precisely to allow distributed migration decisions: when a *SlaveServer* receives an *update-current-status* command, if the *futureVmMap* of its *HostDescriptor* contains a new virtual machine (not currently hosted), the *SlaveServer* can detect from the *VmDescriptor*'s *currentHostId* where it is currently located.

After phase 2, the *SlaveServer* unlocks its strata so that, of sums other *MasterClient* is waiting for the *SlaveServer*'s *HostDescriptor*,it can be sent.

Finally, the *SlaveServer* checks if the changes in his *HostDescriptor* brought the machine into a critical situation, ad as before, if so, it starts a *MasterClient*.

2.3 Policy Layer

The Policy layer (see figure 1) has a main component called *Allocator* which can be extended to implement the specific policy.

To test the local protocol performance, we start analyzing the results of a best fit policy properly customized to define a strict order relation on hosts for each virtual machine of the cloud. We call it Green-DAM policy.

Listing 3: Green-DAM code

```
 1
 2 //Input: VmDescriptor[] vmList,
      HostDescriptor[] hostList.
 3 vmList.sortDecrescent();
 4 foreach vm in vmList{
 5   bestHost = vm.getCurrentHost();
 6   min = bestHost.getAvail(vm)
 7   foreach host in hostList{
 8     double mips = host.getAvail(vm);
 9     if (mips<min || (mips==min &&
10       host.getId()<bestHost.getId())
            ){
11       min = mips;
12       bestHost = host;
13     }
14   }
15   bestHost.allocate(vm);
16 }
```

As in a common best fit policy, the virtual machines are ordered from the one that uses the higher value of CPU MIPS to the lowest, then for each host in the neighborhood we take into account the value of $getAvail(vm)$ (line 8), that is the amount of CPU MIPS that remains available if we allocate the virtual machine *vm*.

The unique variation in respect to a standard best fit policy is in the criterion to select the host: for each virtual machine *vm*, the host with the highest value of $getAvail(vm)$ is chosen as destination of the migration, but if there are two nodes with the same value, the one with the lower id is chosen. Since no physical host is allowed to have the same id of another in the cloud, this simple variation of the best fit policy establishes a strict order relation on nodes at each virtual machine allocation cycle.

This strict order peculiarity of Green-DAM comes from the need to avoid loops in virtual machine migration. We explain the motivations in details in the following subsection.

2.3.1 Policy implementation constraints

The protocol execution allows the system to take a decentralized decision about migrations, but is not able to avoid by itself the possibility that a vm is never-ending migrated through hosts because a node can only have a local knowledge of the neighborhood. The migration policy must be conveniently built to avoid this unpleasant situation.

For example, we consider three partially overlapping neighborhoods:

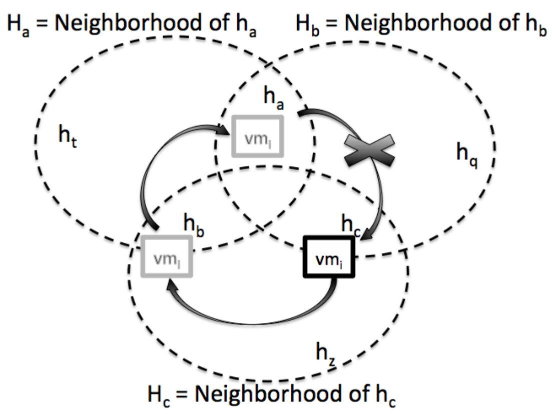
$$H_a = \{h_a, h_b, \dots, h_t\} \tag{2}$$

$$H_b = \{h_b, h_c, \dots, h_q\} \tag{3}$$

$$H_c = \{h_a, h_c, \dots, h_s\} \tag{4}$$

where H_a in is the neighborhood of node h_a, H_b is the neighborhood of node h_b and H_c is the neighborhood of node h_c.

As shown in figure 4, suppose that a protocol execution by the *MasterClient* of h_b decides to migrate a virtual machine vm_i currently allocated on h_c to h_b. When the *SlaveServer* of h_b is unlocked,the policy execution on h_a's *MasterClient* can decide to put vm_i into h_a. Now if h_c has a *MasterClient* running, and decides to migrate vm_i back to h_c, then h_c can take the same decision as before and a loop in vm_i migration starts. If this happens the distributed system will



H_a = Neighborhood of h_a
H_b = Neighborhood of h_b
h_a
h_t
h_q
vm_i
h_b
h_c
vm_i
vm_i
h_z
H_c = Neighborhood of h_c

Figure 4: Example of three overlapping neighborhoods .

never converge to a common decision. Conversely in the following we show that if for each virtual machine the policy can define a chain on possible destination hosts (from the best to migrate in, to the worst one), the distributed execution will converge to a common migration decision.

For each virtual machine, we consider the subset of possible destination hosts $H = \{h_a, h_b, \ldots, h_x\}$. If the chosen policy defines a strict order relation on H, we can define a "better then" operator $>_{\beta(vm_i)}$, such that for virtual machine vm_i, the following relation holds:

$$h_a >_{\beta(vm_i)} h_b >_{\beta(vm_i)} h_c >_{\beta(vm_i)} \cdots >_{\beta(vm_i)} h_x \qquad (5)$$

This means that h_a is the best host of the cloud in which vm_i can be placed. Furthermore we can observe that $>_{\beta(vm_i)}$ operator is transitive for the strict total order definition.

If we consider 3 DAM neighborhoods 2.3.1,3 and 4, as subsets of H, a protocol execution on H_b can migrate vm_i, currently allocated on h_c, to h_b (because $h_b >_{\beta(vm_i)} h_c$) the same on H_a can move it from h_b to h_a (because $h_a >_{\beta(vm_i)} h_b$, but no execution on H_c can decide the migration of vm_i from h_a back to h_c creating a migration loop, because $h_a >_{\beta(vm_i)} h_c$ and the policy only operate following the gradient of the $>_{\beta(vm_i)}$ relation.

Please note that to simplify the scenario we assume a constant load for each virtual machine during the whole simulation. Without this assumption even the total order hypothesis on the policy can not prevent migration cycles.

3 EXPERIMENTAL RESULTS

To understand the efficiency of the proposed model we developed DAM-Sim: a simulator able to apply a specific policy on a collection of neighborhoods through DAM protocol and compare the performance with a centralized implementation of the same policy.

We first tested our approach on a collection of 100 physical nodes hosting 3000 virtual machines (i.e. an average value of 30 virtual machines on each host). We firstly suppose that each node has the MIPS capacity (T_h in section 2.1.1) while the amount of MIPS requested by a virtual machine (m_{vm} in section 2.1.1) is generated random. We fixed the THRESH_DOWN threshold at 60% of computational load, while the TRESH_UP is not used because of our power-saving purposes.

We always start from the worst situation for power-saving purposes, i.e. all the servers are switched on and have the same computational load within the thresholds (OK status). To make the DAM protocol start we forced one node to change its state into UNDER.

In the following graphs, we compare the DAM performance with nN=5, 10, 15, 20 and 25 nodes in each neighborhood, with the application of the centralized policy (Global in figure 5, 6 and 7).

Figure 5 shows the number of servers remained switched on at the end of the execution. As we expected, the DAM protocol cannot perform better than a global algorithm. Indeed the Global best fit policy can always switch off a higher rate of servers resulting in the lower trend of figure 5.

Figure 6 shows the number of migration requested at the end of the simulation. Since no information about the current allocation of a virtual machine is taken into account during the policy computation in a global environment, the number of migrations can be very high. Indeed is high the resulting trend of migration shown by figure6 for Global series, while DAM always outperforms it. This is mainly due to the fact that DAM always ends up with a less power-saving solution, switching off a lower rate of servers. Nevertheless, for high value of computational load the performance of DAM in terms of number of server are comparable to those of the Global best fit policy, while the number of migration requested is significantly lower. Figure 7 shows the number of messages exchanged between hosts during the computation. As we expected it significantly increases as the number of servers in each neighborhood grows. Figure 7 shows that even if the number of message for low values of nN is comparable to the one of the global solution, when the neighborhood dimension grows up the number of messages exchanged significantly increases.

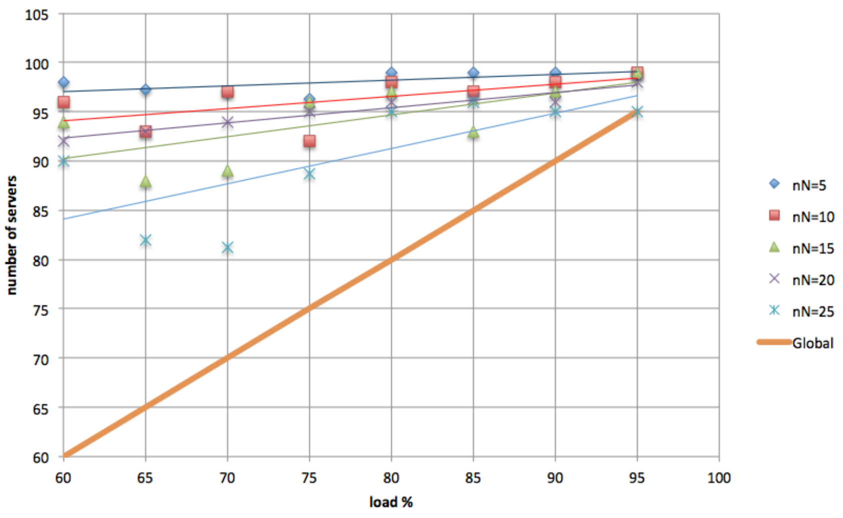


Figure 5: Number of servers switched on after computation

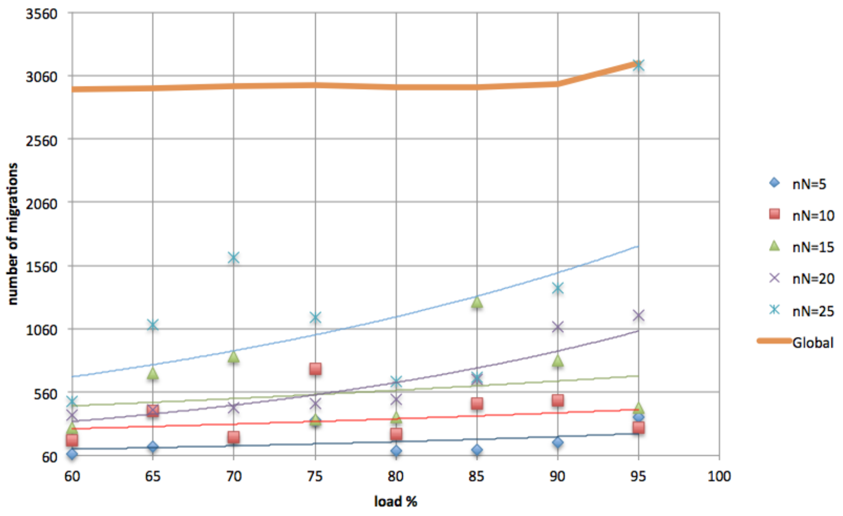


Figure 6: Number of requested migrations

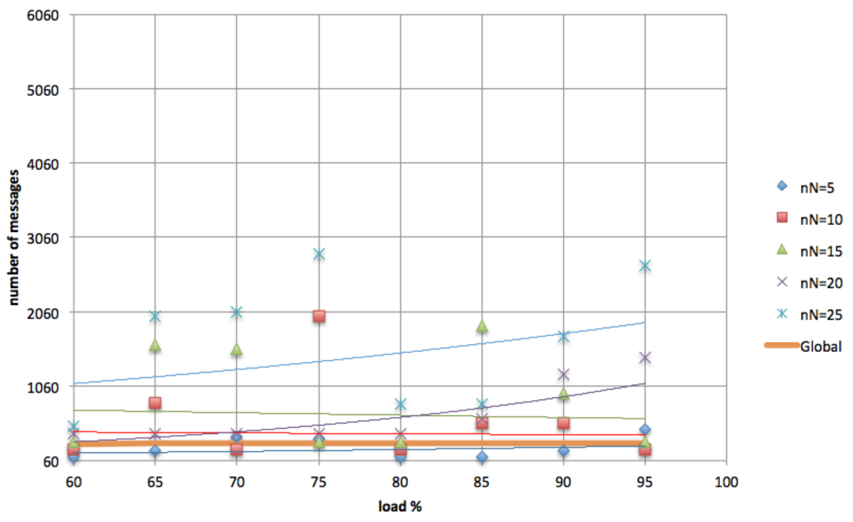


Figure 7: Number of messages exchanged between cloud nodes

4 RELATED WORKS

Our work mainly concern low level infrastructural support, in which the management of virtualized resources is always a compromise between system performance and energy-saving. Indeed in a cloud infrastructure there are usually well-defined SLAs to be compliant to, and perhaps the simplest solution is to use all the machines in the cloud. Nevertheless, if all the hosts of the datacenter are switched on, the energy waist increases leading to probably too high costs for the cloud provider.

Around cloud environments, with their contrasting targets of energy-saving versus performance and SLAs compliance, a lot of work was done in order to provide some kind of autonomy from human system administration and reduce complexity. Some of these works involves automatic control theory realizing an intrinsic centralized environment, in which the rate of utilization of each host is sent to a collector node able to determine which physical machines must be switched off or turned on (Kalyvianaki, 2009; Jung, 2010; Lim et al., 2009). Some other solutions concern centralized energy-aware optimization algorithms (Beloglazov and Buyya, 2012; Jansen, 2011; Younge, 2010), in particular extensions of the Bin Packing Problem (Levine and Ducatelle, 2003; Zaman and Grosu, 2010) to solve both virtual machines allocation and migration problems (Beloglazov et al., 2012). These approaches focus on finding the best solution and minimizing the complexity of the algorithm, without concerning the particular implementation, but assuming a solver aware of the whole system state (in terms of load on each physical host and virtual machine allocation). Thus they particularly lend to a centralized implementation.

Finally, other approaches involve intelligent, optionally bio-inspired (Giordanelli et al., 2012; Balasubramaniam et al., 2006), agent-based system, which can give to the datacenter a certain rate of independence from human administration, showing an intelligent self-organizing emergent behavior (Marzolla et al., 2011; Vichos, 2011), and also provide the benefits of a more distributed system structure.

As in (Marzolla et al., 2011) which is based on Gossip protocol (Jelasity et al., 2005), we adopt a self-organizing approach, where coordination of nodes in small overlapping neighborhoods leads to a global reallocation of virtual machines, but differently from that we created a more elaborate model of communication between physical hosts of the datacenter. In particular while in (Marzolla et al., 2011) each migration decision is taken after a peer-to-peer interaction comparing the states of the only two hosts involved, in our approach the migration decisions are more accurate because they comes from a valuation of the whole neighborhood state.

5 CONCLUSIONS

We presented a distributed solution for cloud virtual infrastructure management in which the hosts of the datacenter are able to self-organize and reach a global virtual machine reallocation according to a given policy. To do so we developed a protocol of communication and tested its behavior with a particular policy by means of a software simulator. We also defined how policies should be built to ensure the protocol convergence. We showed that the protocol has a good behavior in terms of number of migrations requested varying the computational load and the number of server, while as regards power-saving, DAM is still outperformed by a global optimization strategy.

In the near future we will investigate the behavior of our approach with other more complex configurations, where more than one node starts detecting to be in a critical situation. We will also use DAM-Sim to test different and more elaborate reallocation policies, possibly introducing contrasting targets (e.g. power saving with a certain rate of load balancing) and we will take into account other rising conditions, more complex than the simple application of two fixed thresholds (THRESH_UP and TRESH_DOWN).

Finally we would like to test our implementation on a real cloud infrastructure and compare the time to get a common distributed decision with the centralized implementation of the same reallocation policy.

REFERENCES

Balasubramaniam, S., Barrett, K., Donnelly, W., and Meer, S. V. D. (2006). Bio-inspired policy based management (biopbm) for autonomic communications systems. In *7th IEEE workshop on Policies for Distributed Systems and Networks*.

Beloglazov, A., Abawajy, J., and Buyya, R. (2012). Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future Generation Computer Systems*, 28.

Beloglazov, A. and Buyya, R. (2012). Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency Computat.: Pract. Exper.*

Fan, X., Weber, W., and Barroso, L. (2007). Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture ((ISCA 2007)*. ACM New York.

Giordanelli, R., Mastroianni, C., and Meo, M. (2012). Bio-inspired p2p systems: The case of multidimensional overlay. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7.

Jansen, R. (2011). Energy efficient virtual machine allocation in the cloud. In *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE.

Jelasity, M., Montresor, A., and Babaoglu, O. (2005). Gossip-based aggregation in large dinamic networks. *ACM Transaction on Computer Systems*.

Jung (2010). Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *International Conference on Distributed Computing Systems*.

Kalyvianaki, E. (2009). Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In *Proc. of International Conference on Autonomic Computing*.

Levine, J. and Ducatelle, F. (2003). Ant colony optimisation and local search for bin packing and cutting stock problems. *Journal of the Operational Research Society*.

Lim, H. C., Babu, S., and Chase, J. S. (2009). Automated control in cloud computing challeges and opportunities. In *ACDC '09, Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 13–18. ACM New York.

Marzolla, M., Babaoglu, O., and Panzieri, F. (2011). Server consolidation in clouds through gossiping. *Technical Report UBLCS-2011-01.*

Vichos, A. (2011). *Agent-based management of Virtual Machines for Cloud infrastructure*. PhD thesis, School of Informatics, University of Edinburgh.

Younge, A. J. (2010). Efficient resource management for cloud computing environments. In *Green Computing Conference, 2010 International*. IEEE.

Zaman, S. and Grosu, D. (2010). Combinatorial auction-based allocation of virtual machine instances in clouds. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*.