Ant System for Service Deployment in Private and Public Clouds

Máté J. Csorba
Dept. of Telematics,
Norwegian University of
Science and Technology,
N-7491 Trondheim, Norway
Mate.Csorba@item.ntnu.no

Hein Meling
Dept. of Electrical Engineering
and Computer Science,
University of Stavanger,
N-4036 Stavanger, Norway
Hein.Meling@uis.no

Poul E. Heegaard
Dept. of Telematics,
Norwegian University of
Science and Technology,
N-7491 Trondheim, Norway
Poul.Heegaard@item.ntnu.no

ABSTRACT

Large-scale computing platforms that serve thousands or even millions of users through the Internet are on a path to become a pervasive technology available to companies of all sizes. However, existing technologies to enable this kind of scaling are based on a hierarchically managed approach that does not scale equally well. Moreover, existing systems are also not equipped to handle the dynamism that may emerge as a result of severe failures or load surges.

In this paper, we conjecture that using self-organizing techniques for system (re)configuration can improve both the scalability properties of such systems as well as their ability to tolerate churn. Specifically, the paper focuses on deployment of virtual machine images onto physical machines that reside in different parts of the network. The objective is to construct balanced and dependable deployment configurations that are resilient. To accomplish this, a method based on a variant of Ant Colony Optimization is used to find efficient deployment mappings for a large number of virtual machine image replicas that are deployed concurrently. The method is completely decentralized; ants communicate indirectly through pheromone tables located in the nodes.

An example scenario is presented and simulation results are obtained for the method.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Management

Keywords

Deployment, Resource Discovery, Cross-entropy ant system, Cloud computing

1. INTRODUCTION

Cloud computing infrastructures have in recent years become increasingly important for provisioning services that demand reliability and performance, yet are capable to utilize the computing resources efficiently. A major benefit of cloud infrastructures is their ability to dynamically scale up or down as the demand curve changes. One approach in which such dynamic service delivery can be accomplished is through the use of Virtual Machine (VM) images that can be deployed on demand within the cloud. Such VM images are packaged in a standardized way that allows for dynamic deployment, e.g. the Amazon Machine Image format [2]. Moreover, use of a common VM packaging format also enable a computing model where both public and private cloud providers can interoperate. In this context, a *public cloud provider* offers a large infrastructure of compute resources that are provided to (paying) users over the Internet. This is sometimes called Infrastructure-as-a-Service, and Amazon EC2 [2] is an example of a public cloud provider. On the other end, we have *private clouds* that offer a more limited scale of resources, typically accessible only to users directly affiliated with the private cloud owner, e.g. a single organization. These organizations may be running Ubuntu's Enterprise Cloud solution [27], which is compatible with Amazon EC2 in packaging format. The intention of the Eucalyptus project, for example, is to support multiple cloud computing interfaces while preserving the back-end infrastructure [20]. Lack of service management facilities and interoperability between cloud providers have been identified as major obstacles limiting scalability of federated cloud computing environments [22]. Such environments need a unified interface for dynamically managing VMs forming cloud services. Moreover, a heterogeneous cloud computing architecture must also tackle the placement, migration, and monitoring of VMs across interoperability boundaries [10]. In this work however, we rely on the presumed existence of such interoperability and service management facilities, and focus our attention on the *service placement problem.* As such our approach is independent of the specific flavors of the underlying interoperability and management facilities provided.

In this paper, we examine the effects of a hybrid environment in which services are deployed in either the private cloud, public clouds, or both depending on the present usage pattern. Such a scenario is especially interesting with respect to handling load overshoots that may be caused by dependability and/or performance requirements. For example, as the service usage pattern change, VM instances may be added or removed from the public cloud, while retaining the same number of VM instances within the private cloud. During execution in such a hybrid cloud environment, a plethora of highly dynamic parameters influence the optimal deployment configurations, e.g. due to the influence of concurrent services and varying client load. Ideally, the deployment mappings should minimize and balance resource consumption, yet provide sufficient resources to satisfy the dependability requirements of services. However, Fernandez-Baca [11] showed that the general module allocation problem is NP-complete except for certain communication configurations, thus heuristics are required to obtain solutions efficiently.

Our approach is based on a heuristic and decentralized optimization method aimed at *finding suitable mappings* between VM replicas and nodes, in the various clusters of the network, capable of hosting them. The set of mappings selected are constrained in three dimensions ensuring: cloud internal load balancing, cloud global load balancing, and availability of VM replicas in multiple clusters for improved dependability. To accomplish this we use the Cross-Entropy Ant System (CEAS) [12], which is based on Ant Colony Optimization (ACO) [9]. CEAS uses ant-like agents, denoted *ants*, that can move around in the network, identifying potential locations where replicas might be placed.

Related Work.

The notation of regenerating replicas to replace crashed ones was first proposed by Pu [21] in the context of the Eden system. More recent systems [18, 19] provide automatic reconfiguration and regeneration of replicas in the context of group communication systems, and Om [30] focus on regeneration in a peer-to-peer wide-area storage system. Another recent initiative [10] propose similar mechanisms for placement, migration and monitoring of components in the cloud. Such systems provide the underlying mechanisms that are necessary to support service deployment in cloud environments. However, focus is mostly on failure recovery by regeneration of new replicas to improve availability and reliability, and do not try optimize the replica-to-node mappings. Yu and Gibbons [29] show theoretically that replica placements of inter-correlated objects can significantly impact system availability if not placed appropriately. Our work is focused on finding suitable (near optimal) replica-to-node mappings that improve both availability and load balancing properties. Albrecht et al. [1] describe a generic wide-area resource discovery system taking a database-like approach to enable querying for available resources; they use a *centralized* group-finding optimization algorithm to find mappings. Their approach rely on extensive measurement data collection, in some sense not unlike our ants. However in our approach, measurements are not stored centrally in a database or in a DHT. Instead, ants encode such measurement data using a *decentralized* mathematical framework that enable us to heuristically find near optimal solutions rapidly. Many other frameworks have focused on finding optimal placements for virtual machines under a variety of constraints [26, 16]. Maximizing the utility of services via deployment decision making has been investigated in [17]. However, these approaches rely on a centralized optimizer that often has to crawl through the entire state-space of decision alternatives. The SmartFrog [24] deployment and management framework from HP Labs describes services as collections of components and applies a distributed engine comprised of daemons running on every node in a network. Fuzzy learning is applied for configuration management in server environments targeting efficient resource utilization by Xu et al. in [28]. Biologically-inspired resource allocation algorithms in service distribution problems have been targeted by the authors of [14].

Our approach is self-organizing and uses a fully decentralized optimization technique based on the CEAS system [12] which is adaptive to network dynamics and is particularly suited for multi-constrained optimization problems. Our previous work [8] has focused on finding efficient mappings within relatively small scale clusters; and we experimented with different pheromone encodings for improving scalability in [7]. To further study the efficiency of our approach and cross-validate our results against centralized solutions we are also working on mixed integer programs (MIPs) capable of providing optimal replica mappings based on a global view of the system. Similar centralized solutions, in particular integer linear programs, have been applied to clustering problems in grid file systems [25]. Preliminary results of our work to evaluate our approach can be found in [4]. In this paper we extend our approach to consider resource allocation in federated public and private clouds as well as handling the optimal utilization of resources in case of overshoot scenarios.

In the next section we introduce the system model and notation we use, how the deployment rules and the cost function are defined. The basics of the CEAS are presented in Sec. 3, followed by a description of the algorithm we propose in Sec. 4. An example scenario and corresponding simulation results are shown in Sec. 5. Finally, in Sec. 6 we conclude and touch upon future work.

2. SYSTEM MODEL

In this section we introduce the system model and the notation that we use. We also clarify our assumptions, and define dependability constraints and rules related to deployment mapping. Finally, we present a cost function aimed to guide our heuristic search algorithm.

Model and Notation.

We model the system as a large collection, $\mathcal{N}$, of interconnected nodes. $\mathcal{N}$ is partitioned into a set $\mathcal{D}$ of *clusters*, as illustrated by d_1 and d_2 in Fig. 1. Clusters are usually formed according to geographical location or otherwise distinct administrative region. We assume that compute clouds are provided by a single administrative entity that can consist of several clusters. Our objective is to find deployment mappings for this environment for a set of *services*, $\mathcal{S} = \{S_1, S_2, \ldots\}$. Each service may contain replicated VMs to provide the service with fault tolerance and load-balancing. The deployment mapping for S_k is defined as a set of mappings $\mathcal{M} : S_k \to \mathcal{N}$. Let V_i^k be the i^{th} VM of S_k, where $S_k = \{V_1^k, \ldots, V_q^k\}$ is the set of VMs constituting service S_k, q being the number of VMs in the service, $|S_k|$. Accordingly, let R_{ij}^k denote the j^{th} replica of V_i^k so that $V_i^k = \{R_{i1}^k, \ldots, R_{ip_i}^k\}$, where $p_i \geq 1$ is the redundancy level of VM V_i^k. Then, for service S_k, the set of VM instance replicas becomes $S_k = \{R_{11}^k, \ldots, R_{1p_1}^k, \ldots, R_{i1}^k, \ldots, R_{ip_i}^k\}$. In the

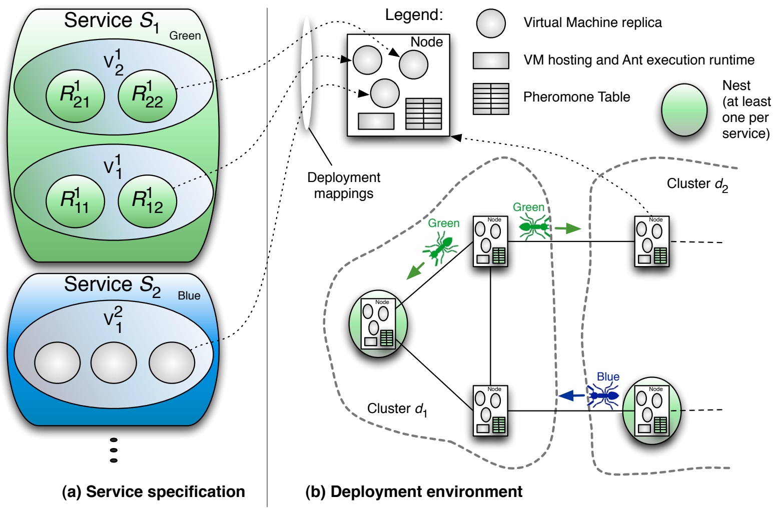

Figure 1: Overview of the deployment environment and service specification.

remainder of the paper, we use the terms VM replica and VM instance interchangeably.

The objective of our deployment logic is to find suitable *mappings* between VM replicas and nodes, capable of hosting them in the various clusters of the network. Using CEAS, ants move around in the network, trying to identify potential locations where replicas might be placed. Ants have associated state; as such they can be implemented as messages on which our algorithm is executed in every node they visit. For each service, there is one *ant species* responsible for finding the deployment mapping for its associated service. This is illustrated in Fig. 1 by the green and blue ant species representing the green and blue service, respectively. It is important to notice that a species of ants corresponds to a service, i.e. a set of VM replicas $S_k = \{R_{11}^k, \ldots\}$. Thus, an increase in the number of VM instances within a service in itself does not lead to impairment of scalability.

As shown in Fig. 1, every node contains an *execution runtime*, that supports installing, running and migrating replicas. Furthermore, each node also has a *pheromone table* that will be updated and read regularly by the ants. The purpose of the pheromone table is to assist ants in selecting suitable deployment mappings; this is in contrast to the original ant system proposed by Dorigo et al. [9], in which pheromones are used for ant routing. For every service that has to be deployed, at least one node must also host an ant *nest*. The tasks of a nest are twofold:

1. to emit ants for its associated service, and

2. trigger installation of VM replicas onto nodes according to found deployment mappings.

Installation is triggered once a predefined *convergence criteria* is reached, e.g. after a certain number of iterations of the algorithm, or when a sufficiently low deployment cost level is reached. The actual installation of the VMs of a service is taken care of by the *execution runtime*, details of which are not discussed here, instead we refer to related work (in Sec. 1). Our goal is to build a core logic for optimizing the deployment mappings and, at the same time, enable compatibility with existing frameworks and interfaces for deployment in the clouds.

An iteration, r, of the algorithm is defined as one round-trip trajectory of the ant. During an iteration r, the ant builds and carries along a *hop list*, H_r, keeping track of the visited nodes. The nest can also be replicated for fault tolerance, thereby emitting ants for the same service from multiple nodes. Synchronizing these nests is not necessary, however, only one designated nest is allowed to trigger physical placement of VMs. In Fig. 1, the green service has two nest replicas.

Mapping Rules.

The CEAS approach is a heuristic optimization method, and as such our target is not to find the globally optimal solution. This is simply because by the time the optimal mapping configuration could be found and installed, it might be suboptimal due to dynamics of the system. Instead, we aim to find a *feasible mapping*, meaning that it satisfies the requirements for the deployment of the service, e.g. in terms of redundancy and load-balancing. Below we will define a set of rules, denoted Φ, to encapsulate these requirements. One of the key functions in CEAS is the use of a *cost function*, denoted $F()$, that evaluates the quality of a mapping M_r found in iteration r of the algorithm. Thus, the objective of the algorithm is to minimize the cost of the mapping $F(M_r)$ subject to Φ. It should be noted that the algorithm continues to optimize the mapping after an appropriate mapping has been found and applied in the network, once a (significantly) better mapping is found, reconfiguration can take place. The dependability rules that constrain the minimization are defined using the two mapping functions, $f_{j,d}$ and g_j, below that apply to service k.

DEFINITION 1. *Let $f_{R,d} : R \to d$ be the mapping of replica R to cluster $d \in \mathcal{D}$.*

DEFINITION 2. *Let $g_{R,n} : R \to n$ be the mapping of replica R to node $n \in \mathcal{N}$.*

Using the two mapping functions defined above we now specify two dependability rules. The first rule, ϕ_1 below, requires replicas to be dispersed over as many clusters as possible, aimed to improve service availability despite potential network partitions between the clusters. Specifically, replicas of VM V_i^k are mapped to different clusters, until all clusters are present in the mapping or all replicas have been mapped to distinct clusters. If the redundancy level of the VM is greater than the number of available clusters in the network, i.e. $|V_i^k| > |\mathcal{D}|$, at least one VM replica is placed in each cluster. Hence, when $j = u$, replicas of V_i^k may be mapped to the same cluster. The second rule, ϕ_2, prohibits two replicas of V_i^k to be placed on the same node, n.

RULE 1. $\phi_1 : f_{R_j,d} \neq f_{R_u,d} \Leftrightarrow (R_j \neq R_u) : \forall d \in \mathcal{D}, \forall R \in V_i^k, \wedge |V_i^k| < |\mathcal{D}|$

RULE 2. $\phi_2 : g_{R_j,n} \neq g_{R_u,n} \Leftrightarrow (j \neq u) : \forall R \in V_i^k$

Combining the rules above we obtain a dependability constraint set for services $\Phi = \phi_1 \wedge \phi_2$. In order to adhere to ϕ_1, the ant gathers data about the clusters utilized for mapping VM replicas; hence, the set of clusters used in iteration r is denoted by D_r. Similarly, the set of replicas from service k mapped to node n in iteration r is denoted by $m_{n,r} \subseteq S_k$. Thus, the ant builds a deployment mapping set $M_r = \{m_{n,r}\}_{\forall n \in H_r}$ for all visited nodes. Finally, ants also collect load-level samples, denoted $l_{n,r}$, from every node $n \in H_r$ visited. The ant uses a *load list*, L_r to carry along all the samples. Load-levels observed by the ant, at the nodes that it visits, are a result of many concurrently executing ant species reserving resources for their respective VM instances. Two different possibilities of implementing this reservation mechanism that serves as a means of indirect communication between ant species have been explored in [5] and [7], here we omit the description of them.

Each VM replica, R_{ij}^k, of a service k has a node-local execution cost (weight of the replica), denoted by $w^k = \{w_{ij}^k\}$, $i = 1 \ldots q, j = 1 \ldots p_i$. This cost is used when ants allocate resources for their corresponding services. Note that, to keep the model simple initially, we consider only identical VM replicas ($w^k = w, \forall i, j, k$). However, in future work we will extend the model to cater for more detailed service models that contain information on individual execution costs for the VM replicas and also communication costs for the communication links between VMs of a service. We have already experimented with these types of costs in previous work in the field of software component deployment [6, 5].

Cost Function.

In what follows, we will define some equations that will be used to define the cost function. Let C_x be a list of values, one for each node visited by an ant. Each value refers to the execution load of the corresponding node.

$$C_x[n] = \left(\sum_{i=0}^{\vartheta_x(n)} \frac{1}{\Theta_x + 1 - i}\right)^2 \quad (1)$$

The algorithm uses two versions of Eq. (1), depending on the parameter $x \in \{0, 1\}$. For $x = 1$, load-level observations, L_r, are used, accounting for all concurrently executing services on the respective nodes. When $x = 0$, the mappings, M_r, made by the ant itself are used, only taking into account the load of those VM instances that are part of the service. The two different usages differ in the upper-bound of the summation and the constant in the denominator, ϑ_x and Θ_x respectively. They are presented next in Eq. (2) and (3).

$$\vartheta_x(n) = |m_{n,r}| \cdot w + x \cdot L_r(n) \quad \text{for } x \in \{0, 1\} \quad (2)$$

$$\Theta_x = \sum_{\forall n \in H_r} \vartheta_x(n) \quad \text{for } x \in \{0, 1\} \quad (3)$$

Θ_x is a constant representing the overall execution load of one service or all services (depending on the parameter x). More specifically, Θ_0 is the total processing resource demand of the service deployed by the ant, whereas $\Theta_1 = \Theta_0 + \mathcal{L}$, where $\mathcal{L}$ represents the additional load of replicas of *other* concurrently executing services. For $\mathcal{L}$, we account only for those instances that are mapped to the nodes visited by the ant, and as such have reserved processing power for themselves. Note that in (3), Θ_x is applied only for the subset of nodes that the ant has visited, H_r. This is favorable for the scalability of the algorithm, since it does not have to explore the entire network $\mathcal{N}$ exhaustively.

With the notational framework in place we are ready to introduce the cost function used to evaluate deployment mappings obtained with CEAS. To take into account the requirements of load-balancing and dependability (according to Φ) when obtaining VM replica mappings the following cost function is defined.

$$F(D_r, M_r, L_r) = \frac{1}{|D_r|} \cdot \sum_{\forall n \in H_r} C_0(n) \cdot \sum_{\forall n \in H_r} C_1(n) \quad (4)$$

Note that, we use (1) to favor globally balanced mappings, i.e. to distribute VM instance load on the network as evenly as possible. In (4) the first term corresponds to enforcing ϕ_1. The second term, C_0 applies solely to the VM replicas of the service the ant is responsible for, thus penalizing the violation of ϕ_2. The last term, C_1, is used for load-balancing and, as such, it takes into account load imposed on nodes by the other services in the network.

Next, we discuss how the CEAS uses the cost function to guide the ants in finding an optimal deployment mapping and we present the definition of pheromone values.

3. CROSS ENTROPY ANT SYSTEM FOR REPLICA DEPLOYMENT

This section describes the basics of the CEAS method necessary for presenting our deployment algorithm.

The core of our deployment logic is built around the CEAS method [12], which can be considered a subclass of ACO algorithms [9]. ACO systems have proven to be able to find the optimum solution to a problem at least once with a probability close to one. Once the optimum has been found, convergence is assured in a finite number of iterations. The logic employs ants searching iteratively for a solution. Solutions found by ants are evaluated using a predefined cost function ($F(M_r)$) that takes into account the constraints of the problem. Every iteration consists of a round-trip by the ant and has two distinct phases. In the first phase, the ant conducts a *forward search* and tries to find a mapping for all VMs in the service it is responsible for. Once a complete mapping has been found, the suggested solution is evaluated using the cost function. In the second phase of the lifetime of an ant, called *backtracking*, ants deposit *pheromone* markings at each node they have visited, much like it is done in the real world when ants forage for food.

The key idea is that these pheromone values are proportional to the quality of the solution, which was determined by the cost function. The optimum is then approached gradually by using the pheromone tables during forward search for selecting VM instance mappings in nodes. Note that, ants have two modes of operation denoted *explorer ants* and *normal ants*. Normal ants behave as described above, using pheromone tables during forward search. On the other hand, explorer ants ignore pheromone markings during forward search; instead they do a random exploration of the search space. The ratio of normal vs explorer ants is configurable; typically 5-10 % are dedicated as explorer ants. The concept of explorers is used two ways, first to detect changes or better opportunities in the environment, and second, to reduce the occurrence of premature convergence leading to sub-optimal solutions.

A cornerstone in CEAS is the *Cross-Entropy* (CE) method proposed by Rubinstein [23]. In CEAS, the CE method is used both to evaluate solutions and for updating pheromone values. Specifically a probability matrix, $\mathbf{p}_r$, is modified gradually according to the cost returned by the cost function $F()$. The objective of applying the CE method is to minimize the cross entropy between consecutive probability matrices $\mathbf{p}_r$ and $\mathbf{p}_{r-1}$. For an introduction and other example applications, see [12].

Let $\tau_{mn,r}$ denote the pheromone value. This value is essentially an encoding of the VM instance mapping $m_{n,r}$ at node n in iteration r. Hence, the pheromone database must be able to store pheromone values that encode the various deployment configurations for various services. Three possible pheromone encoding techniques are discussed and evaluated in [8]; herein the best encoding is used.

While visiting a node, explorer ants select a set of VM replicas to map to that node with the uniform probability $1/|V_i^k|$, where $|V_i^k|$ is the number of replicas to be deployed. On the other hand, normal ants select VM replicas to deploy based on a *random proportional rule*. This rule is encoded as a probability matrix, $\mathbf{p}_r = \{p_{mn,r}\}$.

$$p_{mn,r} = \frac{\tau_{mn,r}}{\sum_{l \in M_r} \tau_{ln,r}} \qquad (5)$$

Updates to the pheromone values are controlled by a *temperature* parameter, γ_r. The temperature is chosen so as to minimize the performance function, $H()$, below.

$$H(F(M_r), \gamma_r) = e^{\frac{-F(M_r)}{\gamma_r}} \qquad (6)$$

$H()$ is applied consecutively to all mappings (*samples*) obtained in all iterations. The expectation of the overall performance then satisfies

$$E_{\mathbf{p}_{r-1}}(H(F(M_r), \gamma_r)) \geq \rho \qquad (7)$$

$E_{\mathbf{p}_{r-1}}(X)$ is the expected value of X s.t. the rules in $\mathbf{p}_{r-1}$; ρ is a *search focus* parameter close to 0 (we use $\rho = 0.01$). Further, the CE method is used to obtain a new set of rules for the next iteration, $\mathbf{p}_r$, by minimizing the cross entropy between two consecutive rules with respect to γ_r and $H(F(M_r), \gamma_r)$. This is achieved by applying the random proportional rule, Eq. (5), for $\forall_{m_n}$ using the pheromone value

$$\tau_{mn,r} = \sum_{k=1}^{r} I(l \in M_r) \beta^{\sum_{j=k+1}^{r} I(j \in M_k)} H(F(M_k), \gamma_r) \qquad (8)$$

where the indicator function $I(x) = 1$ if x is true, or 0 otherwise. For details and proofs of the CE method see [23].

To avoid centralized tables or control, or batches of synchronized iterations, the cost values have to be calculated *immediately* when a new sample (M_r) has been obtained, i.e. in each iteration. To enable this, an auto-regressive version of the performance function is used, as follows

$$h_r(\gamma_r) = \beta h_{r-1}(\gamma_r) + (1 - \beta) H(F(\mathrm{M}_r), \gamma_r) \qquad (9)$$

where $\beta \in \langle 0, 1 \rangle$ is a *memory factor*. β is used to give proper weights to the output of the performance function introduced above. To avoid rapid undesirable changes in the deployment mapping, the performance function will smooth variations in the cost function. The *temperature* parameter, γ_r, is determined by minimization s.t. $h(\gamma) \geq \rho$ (cf. [12])

$$\gamma_r = \{\gamma \mid \frac{1-\beta}{1-\beta^r} \sum_{k=1}^{r} \beta^{r-k} H(F(M_k), \gamma) = \rho\} \qquad (10)$$

To avoid having to store and process the entire mapping history $F(M_r) = \{F(M_1), \ldots, F(M_r)\}$ for each iteration r, which would make the system impractical, we instead assume that subsequent changes in γ_r are relatively small. Hence, we can apply a first order Taylor expansion on Eq. (10). Similarly, for Eq. (8) a second order Taylor expansion can be applied to save memory and processing power [12].

4. VIRTUAL MACHINE REPLICATION IN CLOUD COMPUTING

We have developed an optimization algorithm using CEAS that aims to find suitable deployment configurations for VM replicas in a cloud computing setting. This algorithm is described in this section.

The components of the logic are summarized in the class-diagram in Fig. 2. The mayor component is the $Nest$ that is placed on one of the nodes in the network. It is allowed for one species to have more than one nest for additional dependability. Every node must have some additional properties to support the deployment logic, including an instance of the $PheromoneTable$ that is used as a container for the distributed information needed by the logic, i.e. $\tau_{mn,r}$. A $Nest$ has in addition a set of CEAS related parameters and variables represented by the component $CEASParams$, which information is shared by all the nests of the same species, e.g. β, ρ. In addition, a $Nest$ must be able to access information about the service that the species has to deploy, this component is called the $ServiceRecord$, corresponding to S_k. The second mayor building block, Ant, of the logic is realizing the ants emitted iteratively by the $Nest$. Most of the intelligence is carried by the Ant, represented by methods, as well as some of the variables used during one iteration of the optimization process, e.g. the VM instance mapping set *mapping* corresponding to M_r.

Algorithm 1 Summary of the behavior of $Nest_k$ at any node $n \in \mathcal{N}$

```
1: init();
2: while $r < R$                          {Stopping criteria}
3:     emitAnt(serviceRecord, ... );
4:     $r \leftarrow r + 1$              {Increment iteration counter}
```

In Algorithm 1 the behavior of a nest is summarized briefly.

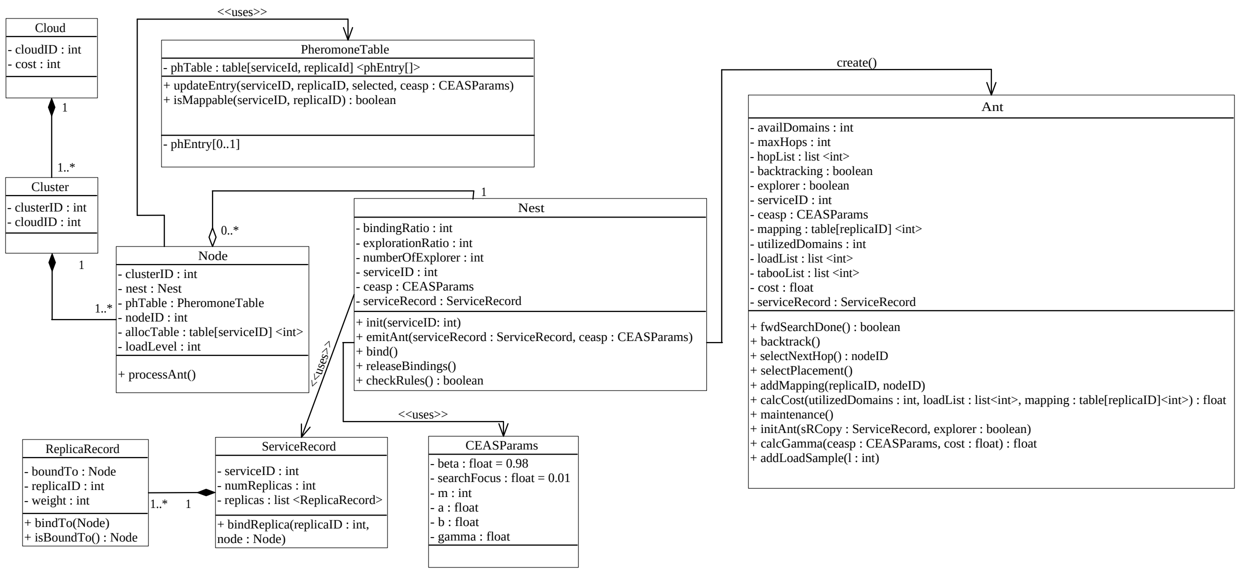

Figure 2: Class Diagram for the CEAS-based Deployment Algorithm

Omitting the details, a $Nest$ emits (creates and resets) $Ants$ sequentially during the optimization process and continues until a stopping criteria, such as a convergence criteria, is fulfilled. Alternatively, nests can continue emitting ants even after the system has stabilized and converged to a given solution, this way providing the capability of adaptation should changes occur in the execution context. Discovery of new, higher utility mappings resulting from context change is supported by explorer ants. Modifying the placement of the services once they are initially deployed can be conditioned by thresholds such as the cost of VM instance migration. Several authors estimate the durations required for migrating operational VMs by conducting experiments. Durations vary, as expected, depending on the hardware context, e.g. bandwidth, and naturally on the VM package size. However, for realistic VM sizes estimates lie typically around 60 to 90 seconds, see [3], [15], [16]. These migration costs can be factored in as threshold values to allow changes in the deployment mappings only if the benefit is higher than the costs of migrating.

The number of iterations required for convergence to an initial stable solution depends on the problem size. For the example scenario, introduced in the next section, the maximum number of iterations allowed for the algorithm was 2000 *explorer* ants followed by an additional 3000 (10% *explorer* and 90% *normal*) ants for each species that were executed in parallel. The algorithm was able to find a solution in all simulation runs within this amount of iterations. Increasing the number of nodes in itself does not make the deployment problem more difficult. An increased network size actually allows to algorithm to find lower cost mappings easier due to the larger amount of available resources. The number of services and the amount replicas within the services is what impacts scalability more, as the number of species executed in parallel is proportional to the number of services and the complexity of one species' task increases as the number of replicas increases [7]. The behavior of an *Ant* is briefly presented in Algorithm 2.

Algorithm 2 Summary of the behavior of a single ant

```
1:  Initialization:
2:      $H_r \leftarrow \emptyset$                      {Hop-list; insertion-ordered set}
3:      $M_r \leftarrow \emptyset$                      {Deployment mapping set}
4:      $D_r \leftarrow \emptyset$                      {Set of utilized domains}
5:      $L_r \leftarrow \emptyset$                      {Set of load samples}
6:  $\gamma_r \leftarrow Nest_k.getTemperature()$       {Get nest temp.}
7:  while not fwdSearchDone()                       {More replicas to map}
8:      $n \leftarrow selectNextNode()$                {Select next node to visit}
9:      if explorerAnt
10:         $m_{n,r} \leftarrow random(\subseteq V_i^l)$     {Randomly select replicas}
11:     else
12:         $m_{n,r} \leftarrow rndProp(\subseteq V_i^l)$    {Select using Eq. (5)}
13:     if $\{m_{n,r}\} \neq \emptyset \wedge n \in d_k$       {Cluster used in mapping}
14:         $D_r \leftarrow D_r \cup d_k$                 {Update utilized clusters}
15:     $M_r \leftarrow M_r \cup \{m_{n,r}\}$
16:     $V_i^l \leftarrow V_i^l - \{m_{n,r}\}$
17:     $L_r \leftarrow L_r \cup \{l_{n,r}\}$          {Estimated proc. load at node n}
18: $cost \leftarrow calcCost(M_r)$                  {Compute cost of mapping}
19: $\gamma_r \leftarrow calcGamma(cost)$             {Compute temp., Eq. (10)}
20: foreach $n \in H_r.reverse()$                  {Backtrack along hop-list}
21:     $n.updatePhTable()$                        {Update pheromones, Eq. (8)}
22: $Nest_k.setTemperature(\gamma_r)$              {Update temp. at $Nest_k$}
```

Every ant that is emitted receives the appropriate parameters from the nest, such as the *explorer* flag, the description of the service to be deployed, CEAS-related parameters, etc. After initialization the *Ant* proceeds with visiting new nodes during *forward search* until the search is done, i.e. a mapping has been found for all the VMs in the service. In other words, the stopping criteria incorporated into the method $fwdSearchDone()$ checks whether the set V_i^l, which is listing the VM instances not yet mapped by the ant during the current iteration, has become empty. Next nodes are selected via the method *selectNextHop*() that takes into account *cluster* taboo-lists and node taboo-lists. Taboo-lists are built by the ant during its *forward search* and are updated continuously adding references to clusters and nodes visited to the two lists respectively. The purposes of the two taboo-lists are to cover all available clusters first, to aid satisfying cluster-disjointness, and if the ant has to proceed even after visiting all the clusters then to avoid revisiting the same nodes. Beside the taboo-lists nodes are selected in a random manner. Mappings at each node are selected by the method $selectPlacement()$ that, depending on whether the ant is an *explorer* or not, uses the local $PheromoneTable$ via Eq. 5 or not. Before leaving the node the Ant also has to sample load-levels ($L_r(n)$) at the node to achieve load-balancing. When *forward search* is done the Ant calculates the cost of the mapping ($F(M_r)$), then recalculates the *temperature* (Eq. 10) and updates the pheromone tables going backward according to its hop-list, H_r, applying Eq. 8. When the ant successfully returns to its nest it is reset and emitted again in the following iteration.

Further improvements in the scalability of the basic approach of CEAS can be made by applying elitism, pheromone sharing and self-tuned packet rate control, additional mechanisms that are described in [13].

5. EXAMPLE SCENARIO AND RESULTS

In this section we present an example cloud computing scenario demonstrating the behavior of our deployment logic. The scenario, as in Fig. 3, consists of 5 private clouds (Cloud $C, \dots, G$) that are connected to the public Internet, thereby enabling connections to public cloud providers (Cloud A, B). Capacities in the public area can thus be utilized on demand, but are subject to economic costs. Conversely usage of the private clusters is free for a service with a home location in that private cloud. Thus, deploying and hosting a VM instance in a node within one of the clouds implies additional costs $|n_i|$, $\forall n_i \in \mathcal{N}$; these costs are summarized in Table 1.

The tangible meaning of the above partitioning and cost assignment is the following concept. It is natural for any organization to execute all VM instances within their privately owned clusters as long as requirements allow, e.g. replication requirements can be satisfied with the available amount of private clusters, as hosting VMs in the private cloud can be considered free compared to costs of the public clouds. In the example setting, there is a trade-off between a large cloud provider with several clusters and plenty of nodes available for placement, which is more expensive to use than paying for hosting in the smaller cloud offering with less resources.

Table 1: Usage costs for the clouds in the example

	Cloud A	Cloud B	Cloud $C \dots G$
Cost $\lvert n_i \rvert$	10	1	0

The deployment task in the case studied in this paper is then to deploy 125 services in total. Administrators in each private cloud have the task of deploying 25 services using their own available resources and, if needed, using public resources as well. It is not allowed, however, to utilize nodes in a private cloud other than the home location for a service (∞ cost for the neighboring private clouds). In this example every service consists of 5 VM instances, among others for dependability reasons, that have to be deployed, thus giving the task of deploying a total number of 625 VMs to the deployment mapping algorithm.

Deployment of the set of VM instances is done in a network environment partitioned into the set of public and private clouds, which in turn are partitioned further into clusters. Every private cloud has two clusters possibly in a private network domain administered by a single authority. In addition the two public clouds Cloud A and Cloud B contain 5 and 3 clusters respectively, resulting in a total of d_i, $i = 1 \dots 18$ clusters. Furthermore, each cluster is a collection of nodes. Clusters in the private clouds consist of 5 nodes, whereas clusters in the public clouds contain 10 nodes each, which gives a total of 130 nodes available in this network.

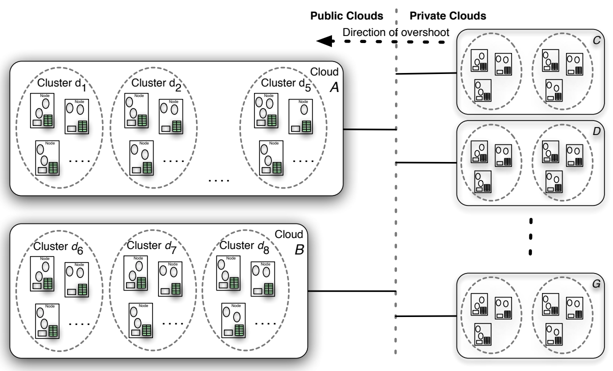

Figure 3: Example scenario

We employ one ant species for each of the services, i.e. there will be exactly 25 ant species for each private cloud responsible for deployment mapping of the 25 services local to the corresponding clouds. In other words, 25 ant nests are placed within every private cloud that execute our algorithm and emit ants accordingly. We define a variable Λ accounting for the additionally incurring costs of using hosts in public clouds as a sum over all hosts that participate in mapping $\mathcal{M}$ obtained during the given iteration

$$\Lambda = \sum_{\forall n_i \in M} |n_i| \tag{11}$$

In our experiment, we executed our algorithm using two extended variants of the cost function. The extension to the original function Eq. (4) is shown in the following

$$F' = F(D_r, M_r, L_r) \cdot (1 + g(x)), \tag{12}$$

where function $g(x)$ is defined in two variants using parameter x and Eq. (11).

$$g(x) = \begin{cases} x \cdot \Lambda, & \text{if linear weighting} \\ 1 - e^{-(x \cdot \Lambda)^2}, & \text{if exponential weighting} \end{cases} \tag{13}$$

To see the resulting VM instance mapping when cloud-related costs are absent we set $x = 0$. On the contrary, to include cloud-related costs the scaling parameter is set to $x > 0$. The exact value of parameter x is dependent on the values we apply as costs of using public clouds, hence it is a scaling parameter. In the example scenario with cloud costs $\{10, 1\}$ the scaling parameter we applied was $x = 0.1$.

The two different alternatives in Eq. (13) represent a linear increment (the former) and an exponential increment (the latter alternative) in cloud costs, when $x > 0$. By applying a more fine-grained exponential weighting to cloud-related costs VM mappings are expected to become more balanced, avoiding under-utilization or overload of clusters.

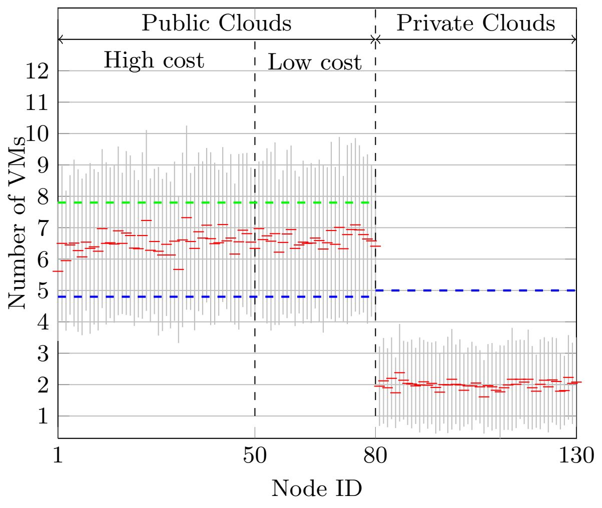

Figure 4: VM instances per node, $x = 0$, no weighting

To investigate the three alternatives we executed simulations of the example setting, running the logic 100 times using each variant of the cost function presented above. In Fig. 4, VM instance mapping is presented in case cloud-related costs are not taken into consideration, i.e. every node has zero cost for hosting a VM. Simulation results are averaged after the algorithm has converged to a solution and deviation from the average number of instances per node is shown as error bars. We can observe that in the first case 2 VMs are mapped on average to hosts within the private clusters, i.e. on nodes $n_{81..90}$, $n_{91..100}$, $n_{101..110}$, $n_{111..120}$, $n_{121..130}$. That means that for the 10 nodes within each private cloud approximately 20 VMs are mapped for hosting, leaving $(5 \cdot 25) - 20 = 105$ VMs for mapping into the public network. Then, as anticipated we have an average around the $(105 \cdot 5)/80 = 6.6$ VMs mapped to the public hosts in the network, which lies between the two extremes of mapping 3 VMs in public and 2 in private clusters $(3 \cdot 125)/80 = 4.7$ and mapping all 5 VMs of a service to public clusters $(5 \cdot 125)/80 = 7.8$, as shown by the horizontal dashed lines. Naturally, the algorithm does not distinguish between the two public cloud offerings in this case.

In the second experiment (Fig. 5) we turned on cloud-related costs and executed our algorithm under the same circumstances otherwise as before. In this case results show that the logic manages to find a mapping that considers the financial penalties of using public clouds. The public cloud with plenty of resources and high cost (nodes $n_{1..50}$) is barely used for deployment, whereas the lower cost public offering is heavily loaded with VMs, while all the dependability requirements are fulfilled, i.e. cluster-disjointness and node-disjointness. At the same time in the private clouds containing 10 nodes, as expected, 5 VMs are mapped to each node on average. This means that each one of the 25 services that are executed within a given private cloud places 1 VM in each of the two local clusters available at 0 cost $((2 \cdot 25)/10 = 5)$. However, due to the cluster-disjointness criteria the 3rd, 4th and 5th VM replica has to be placed to a public cloud with the lowest increment in costs possible, nonetheless taking into account the rest of the requirements.

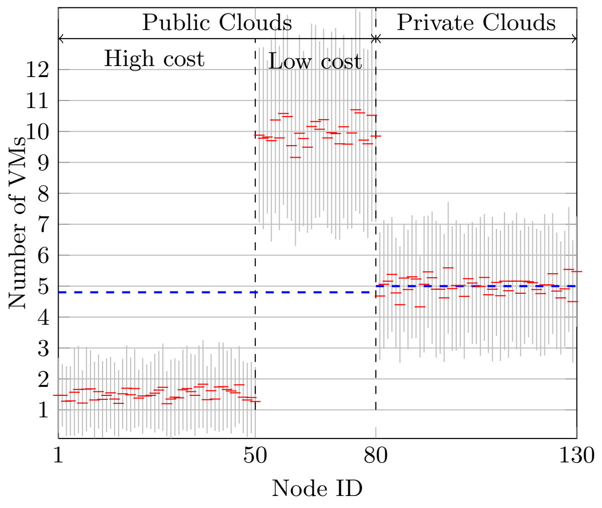

Figure 5: VM instances per node, $x > 0$, linear weighting

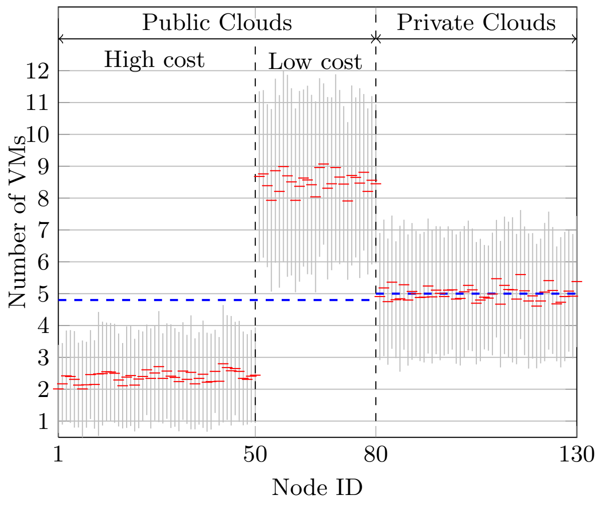

Figure 6: VM instances per node, $x > 0$, exponential weighting

In the third set of simulations (Fig. 6) we executed our deployment mapping algorithm applying the exponential cost function, the second alternative in Eq. (13). Changes from the previous example can mainly be observed in mappings in the public clouds. Mappings in private clouds are not changed due to the application of the same requirements. Using a slightly more complicated cost evaluation in the algorithm, however, we obtained more balanced deployment mappings. Under the given cost values assigned to the different public offerings the cheaper public cloud gets less overloaded with VMs while the number of mappings in the larger public cloud increases to take over some of the execution load while the original requirements remain satisfied.

In the examples above we have shown that the deployment logic we are developing can be applied in a cloud computing scenario by adjusting the corresponding cost functions evaluating VM mappings, thus adapting to different costs related to usage of resources offered by public providers. Using the logic we are able to obtain VM instance mappings that satisfy dependability and performance requirements while minimizing financial penalties in the special case of handling overshoot scenarios in private clouds.

6. CONCLUSIONS

We have presented a swarm intelligence framework targeting the deployment of VM instances in a cloud computing environment. We have designed an algorithm that is fully distributed, scales well by decomposing the problem of deploying multiple services, and paves the way for a deployment logic capable of finding near optimal mappings.

Through the evaluation presented in this paper, we are convinced that our deployment logic is applicable in a cloud computing setting. Nevertheless, we are currently working on a more thorough validation of our approach with new examples. We are also developing a centralized solution using integer linear programming to obtain exact optima as a lower bound for our simulation results. As such, we emphasize the fact that our approach is a heuristic method and may not reach the lower bounds; however, our approach offers the significant advantages of decentralization.

7. REFERENCES

[1] J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson. Design and implementation trade-offs for wide-area resource discovery. *ACM Trans. on Internet Technology*, 8(4), Sept. 2008.

[2] Amazon Elastic Compute Cloud, http://aws.amazon.com/ecs2.

[3] C. Clark et al. Live migration of virtual machines. In *Proc. of the 2nd conf. on Symp. on Networked Systems Design and Implementation - Volume 2*. USENIX Association, 2005.

[4] M. J. Csorba and P. E. Heegaard. Swarm intelligence heuristics for component deployment. In *16th Eunice Int'l Workshop and IFIP WG6.6 Workshop*, Accepted. Springer, Jun 2010.

[5] M. J. Csorba, P. E. Heegaard, and P. Herrmann. Adaptable model-based component deployment guided by artificial ants. In *2nd Int'l Conf. on Autonomic Computing and Communication Systems (Autonomics)*, Sep 2008.

[6] M. J. Csorba, P. E. Heegaard, and P. Herrmann. Cost-efficient deployment of collaborating components. In *8th Int'l Conf. on Distributed Applications and Interoperable Systems*. IFIP, June 2008.

[7] M. J. Csorba, H. Meling, and P. E. Heegaard. Laying pheromone trails for balanced and dependable component mappings. In *4th Int'l Workshop on Self-Organizing Systems*, volume 5918 of *LNCS*, pages 50–64, Zurich, Switzerland, Dec. 2009. IFIP TC 6, Springer-Verlag.

[8] M. J. Csorba, H. Meling, P. E. Heegaard, and P. Herrmann. Foraging for better deployment of replicated service components. In *9th Int'l Conf. on Distributed Applications and Interoperable Systems*, number 5523 in LNCS. Springer-Verlag, June 2009.

[9] M. Dorigo et al. The ant system: Optimization by a colony of cooperating agents. *IEEE Trans. on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1), 1996.

[10] E. Elmroth and L. Larsson. Interfaces for placement, migration, and monitoring of virtual machines in federated clouds. In *8th Int'l Conf. on Grid and Cooperative Computing (GCC 2009)*. IEEE Computer Society Press, Aug 2009.

[11] D. Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Trans. on Software Engineering*, 15(11), 1989.

[12] P. E. Heegaard, B. E. Helvik, and O. J. Wittner. The cross entropy ant system for network path management. *Telektronikk*, 104(01):19–40, 2008.

[13] P. E. Heegaard and O. J. Wittner. Overhead reduction in a distributed path management system. *Computer Networks*, 54(6):1019–1041, 2010.

[14] T. Heimfarth and P. Janacik. Ant based heuristic for os service distribution on adhoc networks. *Biologically Inspired Cooperative Computing*, 2006.

[15] T. Hirofuchi, H. Ogawa, H. Nakada, S. Itoh, and S. Sekiguchi. A live storage migration mechanism over wan for relocatable virtual machine services on clouds. In *9th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid*. IEEE, May 2009.

[16] K. Joshi, M. Hiltunen, and G. Jung. Performance aware regeneration in virtualized multitier applications. In *Workshop on Proactive Failure Avoidance Recovery and Maintenance*, Lisbon, Portugal, June 2009.

[17] R. Kusber, S. Haseloff, and K. David. An approach to autonomic deployment decision making. In *3rd Int'l Workshop on Self-Organizing Systems*, Vienna, Austria, Dec. 2008.

[18] H. Meling and J. L. Gilje. A Distributed Approach to Autonomous Fault Treatment in Spread. In *7th European Dependable Computing Conference*, Kaunas, Lithuania, May 2008. IEEE CS.

[19] H. Meling, A. Montresor, B. E. Helvik, and O. Babaoglu. Jgroup/ARM: a distributed object group platform with autonomous replication management. *Software: Practice and Experience*, 38(9):885–923, July 2008.

[20] D. Nurmi et al. Eucalyptus: an open-source cloud computing infrastructure. *Journal of Physics: Conference Series*, 180, 2009.

[21] C. Pu, J. D. Noe, and A. Proudfoot. Regeneration of replicated objects: A technique and its eden implementation. *IEEE Trans.actions on Software Engineering*, 14(7):936–945, July 1989.

[22] B. Rochwerger et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research & Development*, 53(4), 2009.

[23] R. Y. Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Prob.*, 1999.

[24] R. Sabharwal. Grid infrastructure deployment using smartfrog technology. In *Int'l Conf. on Networking and Services, Santa Clara, USA*, pages 73–79, Jul 2006.

[25] H. Sato, S. Matsuoka, and T. Endo. File clustering based replication algorithm in a grid environment. In *9th IEEE/ACM Int'l Symp. on Cluster Computing and the Grid*. IEEE, May 2009.

[26] A. Verma, P. Ahuja, and A. Neogi. pmapper: power and migration cost aware application placement in virtualized systems. In *9th Int'l Conf. on Middleware*, pages 243–264, Dec. 2008.

[27] S. Wardley, E. Goyer, and N. Barcet. Ubuntu enterprise cloud architecture, Aug 2009.

[28] J. Xu et al. On the use of fuzzy modeling in virtualized data center management. In *Int'l. Conf. on Autonomic Computing*, June 2007.

[29] H. Yu and P. B. Gibbons. Optimal inter-object correlation when replicating for availability. *Distributed Computing*, 21(5):367–384, Feb. 2009.

[30] H. Yu and A. Vahdat. Consistent and automatic replica regeneration. *ACM Trans. on Storage*, 1(1):3–37, Dec. 2004.