Laying pheromone trails for balanced and dependable component mappings

Máté J. Csorba[1], Hein Meling[2], and Poul E. Heegaard[1]

[1] Department of Telematics,
Norwegian University of Science and Technology, N-7491 Trondheim, Norway
{Mate.Csorba, Poul.Heegaard}@item.ntnu.no
[2] Department of Electrical Engineering and Computer Science,
University of Stavanger, N-4036 Stavanger, Norway
Hein.Meling@uis.no

Abstract. This paper presents an optimization framework for finding efficient deployment mappings of replicated service components (to nodes), while accounting for multiple services simultaneously and adhering to non-functional requirements. Currently, we consider load-balancing and dependability requirements. Our approach is based on a variant of Ant Colony Optimization and is completely decentralized, where ants communicate indirectly through pheromone tables in nodes. In this paper, we target scalability; however, existing encoding schemes for the pheromone tables did not scale. Hence, we propose and evaluate three different pheromone encodings. Using the most scalable encoding, we evaluate our approach in a significantly larger system than our previous work. We also evaluate the approach in terms of robustness to network partition failures.

1 Introduction

Data centers are increasingly used to host services over a virtualized infrastructure that permits on-demand resource scaling. Such systems are often comprised of multiple geographically dispersed data center sites to accommodate local demand with appropriate resources, and to ensure availability in case of outages. Major service providers, e.g. Amazon [1], Google, Yahoo! and others all use such infrastructures to power their world wide web offerings, popularly called cloud computing infrastructures. These systems are typically built using large numbers of cheap and less reliable blade servers, racks, hard disks, routers, etc., thus leading to higher failure rate [2]. To cope with increased failure rates, replication and repair mechanisms are absolutely crucial.

Another related and important concern in such data center infrastructures is the problem of *finding optimal deployment mappings for a multitude of services*, while ensuring proper balance between load characteristics and service availability in every infrastructure site. During execution a plethora of parameters can impact the deployment mapping, e.g due to the influence of concurrent services. Another set of parameters in the mix is the dependability requirements of services. Upholding such requirements not only demands replication protocols to ensure consistency, but also adds additional complexity to the optimization problem. Ideally, the deployment mappings should minimize resource consumption, yet provide enough resources to satisfy the dependability requirements of services. However, Fernandez-Baca [3] showed that the general module allocation problem is NP-complete except for certain communication configurations, thus heuristics are required to obtain solutions efficiently.

This paper extends our previous work to find optimal deployment mappings [4], [5] based on a heuristic optimization method called the Cross-Entropy Ant System (CEAS). The strengths of the CEAS method is its capability to account for multiple parameters during the search for optimal deployment mappings [6]. The approach also enables us to perform optimizations in a decentralized manner, where replicated services can be deployed from anywhere within the system, avoiding the need for a centralized control for maintaining information about services and their deployment.

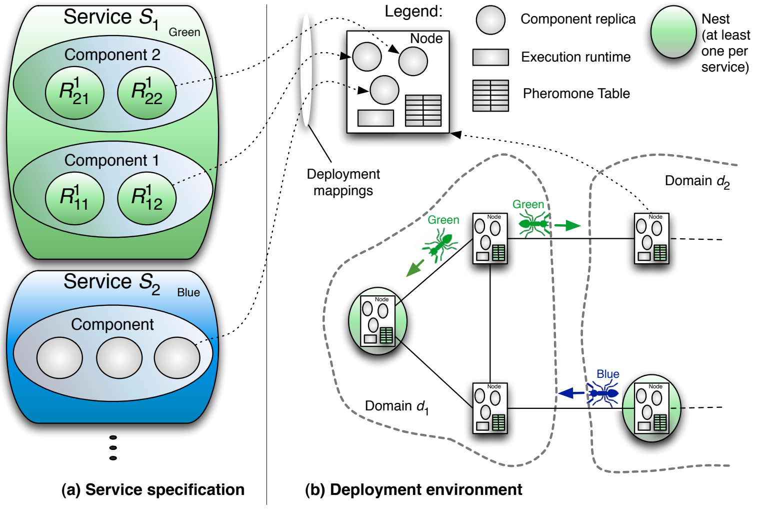
The main goal of this paper is twofold; to provide additional simulation results (i) involving scaling up the problem size, both in terms of number of nodes and replicas deployed, and (ii) evaluating its ability to tolerate network partition failures (split/merge). Scaling up the problem size turned out to be more challenging than first anticipated, and thus certain enhancements were necessary in the algorithm and the data representation. To tackle the challenges we met, we have introduced a new cost function, run-time binding of replicas, a new method for selecting next-hops and new pheromone encodings. In addition, we have used more simple service models in the current study. There are generally two branches of works where finding optimal replica deployment mappings are necessary and useful. On the one hand, virtual machine technology is increasingly being used in data centers for providing high availability and thus needs to consider the placement of replicas in the data centers to ensure efficient utilization of the system resources. The advantage of this approach is that (server) applications running on virtual machines can be repaired simply by regenerating them in another physical machine. This is the approach taken by the Amazon EC2 system [1] and in VMware, among others. The general drawback with virtualization for fault tolerance and high availability is that the storage system used to maintain application state must be independently replicated as it would otherwise constitute a single point of failure. On the other hand, server applications written specifically for fault tolerance typically replicate their application state to all replica processes, avoiding any single points of failure. These systems are typically built using a middleware based on a group communication system with support for repair mechanisms, e.g. DARM [7].

The importance and utility of deployment decision making and optimization has been identified previously, e.g. in [8]. Recently, Joshi et al [9] proposed a centralized approach in which an optimizer and model solver component is used to find optimal mappings specifically in the field of virtual machine technology. We however, intend to pursue a fully distributed solution that is based on optimization techniques and can support context-awareness and adaptation.

The paper is organized as follows. The next section presents our view on component replicas, their deployment, corresponding costs and requirements. In Sec. 3 the fundamentals of the CEAS are described. Sec. 4 proposes our algorithm for solving the deployment mapping problem and subsequently we demonstrate its operation in Sec. 5. Finally, we conclude and touch upon future work.

2 System Model, Assumptions and Notation

We consider a large-scale distributed system consisting of a collection of *nodes*, $\mathcal{N}$, connected through a network. Nodes are organized into a set of *domains*, $\mathcal{D}$, as illustrated by d_1 and d_2 in Fig. 1. All nodes within a domain are located at the same geographic site, whereas different domains are in separate sites. The objective of the distributed system is to provide an environment for hosting a set of *services*, $\mathcal{S} = \{S_1, S_2, \ldots\}$, to be provided to external clients. Let C_i^k be the i^{th} *component* of service k, and let $S_k = \{C_1^k, \ldots, C_q^k\}$ denote the set of components for service k, where $q = |S_k|$. Each component may be replicated for fault tolerance and/or load-balancing purposes. Thus, let R_{ij}^k denote the j^{th} replica of C_i^k. Hence, $C_i^k = \{R_{i1}^k, \ldots, R_{ip_i}^k\}$, where $p_i \geq 1$ is the redundancy level of C_i^k. Moreover, $S_k = \{R_{11}^k, \ldots, R_{1p_1}^k, \ldots, R_{i1}^k, \ldots, R_{ip_i}^k\}$ is the expansion of the component sets into replicas for service k.



(a) Service specification **(b) Deployment environment**

Fig. 1. Overview of the deployment environment and service specification.

The objective of the algorithms herein is to discover suitable *deployment mappings* between *component replicas* (*replicas* for brevity) and nodes in the network, such that the dependability requirements of all services are preserved with minimal resource consumption. To accomplish this, the CEAS optimization method is used, which works by evaluating a *cost function*, $F()$, for different deployment mappings. The CEAS method is implemented in the form of *ants* moving around in the network to identify potential locations where replicas might be placed. An ant is simply an agent with associated state; as such it is simply a message on which the ant algorithm is executed at each visited node. We say that different *ant species* are responsible for different services, e.g. the green and blue ants in Fig. 1 represent the green and blue services, respectively.

As Fig. 1 shows, each node contains an *execution runtime* whose tasks are to install, run and migrate replicas. A node also has a *pheromone table* which is manipulated by ants visiting the node to reflect their knowledge of existing mappings. Moreover, the pheromone table is used by ants for selecting suitable deployment mappings; it is not used for ant routing as in the Ant Colony Optimization (ACO) approach [10]. See Sec. 4.2 for details.

To deploy a service, at least one node must be running a *nest* for that service. The tasks of a nest are twofold: (i) to emit ants for its associated service, and (ii) trigger installation of replicas at nodes, once a predefined *convergence criteria* is satisfied, e.g. after a certain number of iterations of the algorithm. An iteration is defined as one round-trip trajectory of the ant, during which it builds a *hop list*, H_r, of visited nodes. A nest may be replicated for fault tolerance, and emit ants independently for the same service. During execution of the CEAS method, synchronization between nests is not necessary, but only a primary nest will execute deployment decisions. Fig. 1 shows a two-way replicated nest for the green service; nests for the blue service are not shown.

Initially, the composition of services to be deployed is specified as UML collaborations embellished with non-functional requirements that are used as input to the cost function of our algorithm to evaluate deployment mappings (cf. [11]). Our aim is not to find the globally optimal solution. The rationale for this is simple; by the time the optimal deployment mapping could be applied, it is likely to be suboptimal due to dynamics of the system. Rather, 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. These requirements are specified as a set of rules, denoted Φ. Thus, our objective function becomes $min\ F()$ subject to Φ. Moreover, our algorithm can continue to optimize even though an appropriate mapping has been found and deployed into the network. Once a (significantly) better mapping is found, reconfiguration can take place.

Next we define the dependability rules, Φ, that constrain the minimization problem, but first we define two mapping functions. These rules and functions apply to service k.

DEFINITION 1 *Let $f_{j,d} : R_{ij}^k \rightarrow d$ be the mapping of replica R_{ij}^k to domain $d \in \mathcal{D}$.*

DEFINITION 2 *Let $g_j : R_{ij}^k \rightarrow n$ be the mapping of replica R_{ij}^k to node $n \in \mathcal{N}$.*

Rule ϕ_1 requires replicas to be dispersed over as many domains as possible, aimed to improve service availability despite potential network partitions[1]. Specifically, replicas of component C_i^k shall be placed in different domains, until all domains are used. If there are more replicas than domains, i.e. $|C_i^k| > |\mathcal{D}|$, at least one replica shall be placed in each domain. The second rule, ϕ_2, prohibits two replicas of C_i^k to be placed on the same node, n.

RULE 1 $\phi_1 : \forall d \in \mathcal{D}, \forall R_{ij}^k \in C_i^k : f_{j,d} \neq f_{u,d} \ \Leftrightarrow (j \neq u) \wedge |C_i^k| < |\mathcal{D}|$

RULE 2 $\phi_2 : \forall R_{ij}^k \in C_i^k : g_j \neq g_u \ \Leftrightarrow (j \neq u)$

Combining these rules gives us the desired set of dependability rules, $\Phi = \phi_1 \wedge \phi_2$. In order to adhere to ϕ_1, the ant gathers data about domains utilized for mapping replicas; hence, let D_r denote the set of domains used in iteration r. The ant also collects information about replicas mapped to various nodes. Thus, we introduce $m_{n,r} \subseteq S_k$ as the set of service k replicas mapped to node n in iteration r. Moreover, let $M_r = \{m_{n,r}\}_{\forall n \in H_r}$ be the deployment mapping set at iteration r for all visited nodes. Finally, ants also collect load-level samples, $l_{n,r}$, from every node $n \in H_r$ visited in iteration r; these samples are added to the *load list*, L_r, indexed by the node identifier, n.

[1] We assume network partitions are more likely to occur between domain boundaries.

The load-levels observed by an ant are a result of many concurrently executing ant species reserving resources for their respective services. For simplicity, all replicas have the same node-local execution cost, w, whereas communication costs are ignored. An ant during its visit to node n reserves processing resources for the replicas, if any, that it has chosen to map to n. Mappings made at n during iteration r are stored in $m_{n,r}$, thus, resources of size $|m_{n,r}| \cdot w$ are reserved during a visit, assuming identical cost for all replicas. With this notational framework in place, we are now ready to introduce the cost function used by the deployment logic.

First, we define a list, NC_x, that can carry an element for each node visited by the ant and which elements account for specific execution costs imposed on those nodes. Elements of the list are calculated two different ways ($x = 1$ or 2), using the observations on the services executed in parallel (L_r), and the mappings of replicas made by the ant itself (M_r).

$$NC_x[n] = (\sum_{i=0}^{\vartheta_x} \frac{1}{\Theta_x + 1 - i})^2 \tag{1}$$

Parametrization of the list is done by changing the upper bound of the summation, ϑ_x and the constant in the denominator, Θ_x. Accordingly, ϑ_x and Θ_x are defined as follows.

$$\vartheta_x = \begin{cases} |m_{n,r}| \cdot w, & x = 1 \\ |m_{n,r}| \cdot w + L_r(n), & x = 2 \end{cases} \tag{2}$$

The constant, Θ_x represents the overall execution load of one service or all services. In other words, Θ_1 is the total processing resource demand of the service deployed by the ant, whereas Θ_2 represents the overall joint load of the service being deployed and the load of replicas executed in parallel.

$$\Theta_x = \begin{cases} \sum_{\forall n \in H_r} |m_{n,r}| \cdot w, & x = 1 \\ \sum_{\forall n \in H_r} (|m_{n,r}| \cdot w + L_r(n)), & x = 2 \end{cases} \tag{3}$$

Importantly, the equations only have to be applied on the subset of nodes an ant has actually visited (H_r), which is beneficial for scalability as there is no need for exploring the total amount of available nodes. Finally, to build a cost function that satisfies our requirements with regard to Φ, while maintaining load-balancing, we formulate $F()$ using a combination of terms, as shown in (4).

$$F(D_r, M_r, L_r) = \frac{1}{|D_r|} \cdot \sum_{\forall n \in H_r} NC_1(n) \cdot \sum_{\forall n \in H_r} NC_2(n) \tag{4}$$

Thus, we use (1) for load-balancing, i.e. to distribute replicas to the largest extent possible. The three terms correspond to our goals in the optimization process. The first reciprocal term caters for ϕ_1. Applying (1) solely on the replicas of the service the ant species is responsible for ($x = 1$) penalizes violation of ϕ_2, i.e. favors a mapping where replicas are not collocated, but distributed evenly. Lastly, the standard application of (1), $x = 2$, balances the load taking into account the presence of other services during the deployment mapping. A more detailed introduction to the application of the load-balancing term can be found in [5] and [11]. The next section describes how the cost function plays a role in driving the optimization using the CEAS.

3 The Cross-Entropy Ant System

We build our algorithm around the CEAS to obtain optimal deployment mappings with high confidence. CEAS can be considered as a subclass of ACO algorithms [10], which have been proven to be able to find the optimum at least once with probability close to one; once the optimum has been found, convergence is assured within a finite number of iterations. The key idea is to have many ants, search iteratively for a solution according to a cost function defined according to problem constraints. Each iteration is divided into two phases. Ants conduct *forward search* until all the replicas are mapped successfully. After that, the solution is evaluated using the cost function, the ants continue with *backtracking* leaving *pheromone* markings at nodes. This resembles real-world ants foraging for food. The pheromone values are proportional to the solution quality determined by the cost function. These pheromone markings are distributed to nodes in the network, and are used during forward search to select replica sets for deployment mapping, gradually approaching the lowest cost solution. In forward search, a certain proportion of ants do a random *exploration* of the state space, ignoring the pheromone trails. Exploration reduces the occurrence of premature convergence leading to suboptimal solutions. The CEAS uses the *Cross-Entropy (CE) method* introduced by Rubinstein [12] to evaluate solutions and update the pheromones. The CE method is applied to gradually change a probability matrix $\mathbf{p}_r$ according to the cost of the mappings with the objective of minimizing the cross entropy between two consecutive probability matrices $\mathbf{p}_r$ and $\mathbf{p}_{r-1}$. The method itself has been successfully applied in different fields of network and path management, for examples and an intuitive introduction we refer to [6].

In our algorithm, the CEAS is applied to obtain an appropriate deployment mapping, $\mathcal{M} : C_i^k \rightarrow \mathcal{N}$, of the replicas ($C_i^k$) of service S_k onto a set of nodes. A deployment mapping is evaluated by applying the cost function as $F(M_r)$. In the following, let $\tau_{mn,r}$ be the pheromone value corresponding to, $m_{n,r}$, the set of replicas mapped to node n in iteration r. Various pheromone encoding schemes are discussed in Sec. 4.2.

To select a set of replicas to map to a given node, ants use the so-called *random proportional rule* matrix, $\mathbf{p}_r = \{p_{mn,r}\}$ presented below. Similarly, *explorer ants* select a set of replicas with uniform probability $1/|C_i^k|$, where $|C_i^k|$ is the number of replicas to be deployed.

$$p_{mn,r} = \frac{\tau_{mn,r}}{\sum_{l \in M_{n,r}} \tau_{ln,r}} \tag{5}$$

A parameter γ_r denoted the *temperature*, controls the update of the pheromone values and is chosen to minimize the performance function, which has the following form

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

and is applied to all r samples. The expected overall performance satisfies the equation

$$h(p_{mn,r}, \gamma_r) = E_{\mathbf{p}_{r-1}}(H(F(M_r), \gamma_r)) \geq \rho \tag{7}$$

$E_{\mathbf{p}_{r-1}}(X)$ is the expected value of X s.t. the rules in $\mathbf{p}_{r-1}$, and ρ is a parameter (denoted *search focus*) close to 0 (in our examples 0.01). Finally, a new updated set of rules, $\mathbf{p}_r$, is determined by minimizing the cross entropy between $\mathbf{p}_{r-1}$ and $\mathbf{p}_r$ with respect to γ_r and $H(F(M_r), \gamma_t)$. Minimized cross entropy is achieved by applying the random proportional rule in (5) for $\forall_{mn}$ with

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

where $I(x) = 1$ if x is true, 0 otherwise. See [12] for further details and proof.

As we target a distributed algorithm that does not rely on centralized tables or control, neither on batches of synchronized iterations, the cost values obtained by applying Eq. (4) are calculated *immediately* after each sample, i.e. in each iteration r. Then, an auto-regressive performance function, $h_r(\gamma_r) = \beta h_{r-1}(\gamma_r) + (1 - \beta) H(F(\mathrm{M}_r), \gamma_r)$ is applied, where $\beta \in < 0, 1 >$ is a *memory factor* that gives weights to the output of the performance function. The performance function smoothes variations in the cost function and helps avoiding undesirable rapid changes in the deployment mappings.

The *temperature* required for the CEAS, e.g. in Eq. 6, is determined by minimizing it subject to $h(\gamma) \geq \rho$ (cf. [13])

$$\gamma_r = \{\gamma \mid \frac{1-\beta}{1-\beta^r} \sum_{i=1}^{r} \beta^{r-i} H(F(M_i), \gamma) = \rho\} \tag{9}$$

However, (9) is a complicated function that is storage and processing intensive since all observations up to the current sample, i.e. the entire mapping cost history $F(M_r) = \{F(M_1), \ldots, F(M_r)\}$ must be stored, and weights for all observations have to be recalculated. This would be an impractical burden to on-line execution of the logic. Instead, given that β is close to 1, it is assumed that changes in γ_r are relatively small in subsequent iterations, which enables a first order Taylor expansion of (9), and a second order Taylor expansion of (8), see [13], thus saving memory and processing power.

4 Ant Species Mapping Replicas

In this section we present our deployment algorithm, how we apply the CEAS method, and three different ways of encoding replica mappings into pheromone values.

4.1 Swarm-based Component Deployment

Our algorithm has successfully been applied for obtaining component mappings that satisfy non-functional requirements. In addition, the algorithm's capability to adapt to changing network conditions, for example caused by node-failures, has been investigated, cf. [5]. However, from a dependability point of view it is interesting to equip the logic with the capability to adapt to dynamicity of domains, i.e. splitting/merging of domains. To also cater for domain splits and merges we propose to initiate an ant-nest in multiple nodes belonging to separate domains. These ant-nests will emit ants corresponding to the same set of services, this however, will not result in flooding the network with ants as the rate of emission in a stable network can be divided equally between the nests. Besides, ants emitted from different nests but optimizing mappings for the same service will update the same pheromone tables in the nodes they visit during their search for a solution. The concept of multiple nests has been introduced in [11].

Algorithm 1 shows the code of a single ant-nest that sends out an ant in every iteration. The idea is that when a coherent network suffers a split, there shall be at least one nest in each region after the split event that will maintain a pheromone database in each region. (By a region we denote a set of nodes partitioned into one or more domains.)

To ease convergence of the mappings made by the ants the nests are allowed to bind one replica at a time if some condition applies. Here we check rules ϕ_1 and ϕ_2 against the mapping obtained in the current iteration, M_r. Replica bindings are indicated in the service specification that is derived from the model of the service.

Algorithm 1 Code for $Nest_k$ corresponding to service S_l at any node $n \in \mathcal{N}$

```
Initialization:
    $r \leftarrow 0$                                        {Number of iterations}
    $\gamma_r \leftarrow 0$                                        {Temperature}

while $r < R$                                          {Stopping criteria}
    $M_r \leftarrow antAlgo(r, k)$                          {Emit new ant, obtain $M_r$}
    $update(availableDomains)$            {Check the number of available domains}
    if $splitDetected() \vee mergeDetected()$
        $release(S_l)$               {Delete existing bindings for all replicas $c_i \in C_i^l$}
    if $\phi_1(M_r, availableDomains) \wedge \phi_2(M_r)$
        $bind1(M_r)$                 {Bind one of the still unbound replicas in $C_i^l$}
    $r \leftarrow r + 1$                                  {Increment iteration counter}
```

After a replica has been bound to a specific host ants in subsequent iterations will not try to find a new mapping for it, instead these bound mappings are maintained and the search is conducted for the remaining replicas only. Importantly however, bound replicas are also taken into account when the cost of the total mapping is evaluated by the ant. When a split or a merge event occurs these soft-bindings are flushed by the ant nest and, for example in case of a merge, two nests being in the same region can start to cooperate and share bindings and pheromone tables again.

Here it is important to clearly distinguish between the notions of replica mapping, binding and deployment. We use the term mapping during the optimization process, where our algorithm is constantly optimizing an ordering of replicas of a service to underlying execution hosts, but only internally to the algorithm itself. When a replica is bound to a host it means that from that point the algorithm does not change the mapping between that replica and a host. By deployment however, we refer to the actual physical placement of a software component replica to a node, which is triggered after the mappings obtained by our algorithm have converged to a satisfactory solution. The latter property ensures that there is no undesirable fluctuation in the migration of replicas using our method. In Algorithm 2 we present the steps executed by the ants emitted from a nest.

Each species of ants retrieves and updates the temperature used in the CEAS method from the nest where they are emitted from. First, an ant visits the nodes, if any, that already have a bound replica mapped to maintain these mappings, which will be taken into account when the cost of the total mapping is evaluated. The pheromones corresponding to these bound mappings will also be updated during *backtracking*. Besides, ants allocate processing power corresponding to the execution costs of the bound replicas, derived from the service specification. After maintenance the ants jump over to nodes selected in a guided random manner and attempt to map some replicas to the node they reside in. This selection of the next node to visit, in contrast to e.g. ant-based routing algorithms, is independent from the pheromone markings laid by the ants. The selection of replica mappings in each node, however, is influenced by the pheromones.

Here, we distinguish between *explorer* and *normal* ants, where the former selects a set of replicas to map randomly and the latter uses the pheromone table at the current node. In case of a $normal$ ant the selection process varies depending on the form of the pheromone tables (cf. Sec. 4.2). After some variables carried along by the ant (M_r, D_r, C_i^k) are updated a sample of the sum of execution load on the current node is taken by the ant. This replica load reservation mechanism is intended to function as an indirect way of communication between species executed in parallel. At the end of the $forward$ $search$ phase, when the ant has managed to map all the replicas of the service, the mapping is evaluated using the cost function and the temperature is recalculated using the obtained cost value. The last part in the lifetime of a single ant is the $backtracking$ phase, during which the ant revisits the nodes that have been used for the mapping of the service and updates the pheromone database.

Algorithm 2 Ant code for mapping of replicas $C_i^l \in S_l \subset \mathcal{S}$ from $Nest_k$

```
Initialization:
  $H_r \leftarrow \emptyset$                                   {Hop-list; insertion-ordered set}
  $M_r \leftarrow \emptyset$                                   {Deployment mapping set}
  $D_r \leftarrow \emptyset$                                   {Set of utilized domains}
  $L_r \leftarrow \emptyset$                                   {Set of load samples}
function $antAlgo(r, k)$
  $\gamma_r \leftarrow Nest_k.getTemperature()$              {Read the current temperature}
  foreach $c_i \in C_i^l$                                    {Maintain bound replica mappings}
    if $c_i.bound()$
      $n \leftarrow c_i.boundTo()$                           {Jump to the node where this comp. is bound}
      $n.reallocProcLoad(S_k, w)$                            {Allocate processing power needed by comp.}
      $l_{n,r} \leftarrow n.getEstProcLoad()$                {Get the estimated processing load at node $n$}
      $L_r \leftarrow L_r \cup \{l_{n,r}\}$                  {Add to the list of samples}
  while $C_i^l \neq \emptyset$                               {More replicas to map}
    $n \leftarrow selectNextNode()$                          {Select next node to visit}
    if explorerAnt
      $m_{n,r} \leftarrow random(\subseteq C_i^l)$           {Explorer ant; randomly select a set of replicas}
    else
      $m_{n,r} \leftarrow rndProp(\subseteq C_i^l)$          {Normal ant; select replicas according to Eq. (5)}
    if $\{m_{n,r}\} \neq \emptyset, n \in d_k$               {At least one replica mapped to this domain}
      $D_r \leftarrow D_r \cup d_k$                          {Update the set of domains utilized}
    $M_r \leftarrow M_r \cup \{m_{n,r}\}$                    {Update the ant's deployment mapping set}
    $C_i^l \leftarrow C_i^l - \{m_{n,r}\}$                   {Update the set of replicas to be deployed}
    $l_{n,r} \leftarrow n.getEstProcLoad()$                  {Get the estimated processing load at node $n$}
    $L_r \leftarrow L_r \cup \{l_{n,r}\}$                    {Add to the list of samples}
  $cost \leftarrow F(M_r, D_r, L_r)$                         {Calculate the cost of this given mapping, using Eq. (4)}
  $\gamma_r \leftarrow updateTemp(cost)$                     {Given cost, recalculate temperature according to Eq. (9)}
  foreach $n \in H_r.reverse()$                              {Backtrack along the hop-list}
    $n.updatePheromone(m_{n,r}, \gamma_r)$                   {Update pheromone table in n, Eq. (8)}
  $Nest_k.setTemperature(\gamma_r)$                          {Update the temperature at $Nest_k$}
```

The gain in using a guided but random hop-selection instead of a pure random walk lies in that with the proper guidance the frequency of finding an efficient mapping is greater. The idea is that at first the next node is selected from a domain that has not yet been utilized until all visible domains are covered, leading to better satisfaction of ϕ_1. Then the next hop selection continues with drawing destinations from the set of nodes not yet used in the mapping by checking with the variable M_r, before reverting to totally random drawing. The guided hopping strategy for the selection of a next node to visit is summarized in Algorithm 3.

Algorithm 3 Procedure to select the next hop for an ant

```
function $selectNextNode()$                      {Guided random hop}
  if $H_r = \mathbf{N}$                          {All nodes visited}
    $n \leftarrow random(\mathbf{N})$            {Select candidate node at random}
  else
    if $D_r = \mathbf{D}$                        {All available domains utilized}
      $n \leftarrow random(N \setminus M_r)$     {Select a node that has not been used yet}
    else
      $d_i \leftarrow random(\mathbf{D} \setminus D_r)$   {Select a domain not yet used}
      $n \leftarrow random(d_i)$                 {Select a node within this domain}
  $H_r \leftarrow H_r \cup \{n\}$                {Add node to the hop-list}
  return $n$
```

4.2 Encoding Sets of Replicas into Pheromone Entries

Generally, pheromone entries can be viewed as a distributed database located in the nodes available in the network considered for deployment. This distributed database has to be built so that it is able to describe arbitrary combinations of replicas of a given service component. At the same time the size of this database is crucial for obtaining better scalability for our approach. The reasons are twofold. The first reason is related to memory consumption as each participating node has to cater for a pheromone database for each service being deployed. Thus, memory consumption grows with the database size (depending on the encoding) and with the number of parallel services, where we can influence the former. Second, as we can see in the algorithm description in Sec. 4.1, an individual ant agent has to browse through the pheromone entries during its visit at a node, so clearly, a more compact encoding helps speeding up execution of the tasks an ant has to perform. The different encodings we proposed are shown in Table 1.

Table 1. Three pheromone encodings for a service with $|C_i^k|$ replicas

Encoding	DB size in a node	Encoding example w/ $\|C_i^k\| = 4$
bitstring	$2^{\|C_i^k\|}$	$[0000]b \ldots [1111]b$
per comp.	$2 \cdot \|C_i^k\|$	[0/1]; [0/1]; [0/1]; [0/1]
# replicas	$\|C_i^k\| + 1$	$[0] \ldots [4]$

The $bitstring$ encoding is the largest as it has a single value for all possible combinations of replica mappings in every node, which results in prohibitively large memory need. For example, in case of 20 replicas per service this encoding leads to 2^{20} pheromone values, which by using 4 byte long floating point numbers would require 4 MB of memory for each of such services at every node. To tackle this problem we might reduce the pheromone table size by applying more simple bookkeeping taking into account solely the number of replicas mapped to a given node ($\#\ replicas$). This results in the most compact pheromone database, however it comes with a drawback that it can only be applied if there is no need to distinguish between replicas in the service specification (for example considering replication and dependability aspects only). As a trade-off we developed a third encoding ($per\ comp.$) that results in no information loss and still linear growth of the pheromone database. $per\ comp.$ uses one distinct pheromone entry for every replica instance indicating whether or not to deploy a replica at a given node. The drawback is that an ant arriving at a node has to decide on the deployment mapping of each replica, one-by-one reading the multiple pheromone entries. Nevertheless, a reduction in the database structure size is necessary for scaling the algorithm up to larger amounts of nodes and replicas. How the various encodings perform will be demonstrated with an example in Sec. 5.

5 Simulation Results

To evaluate the deployment mapping logic proposed above we start with an example where 10 services ($S_1 \ldots S_{10}$) are being deployed simultaneously, that means 10 independent species are released. Besides, we apply 20 ant nests to look at a simple split/merge scenario where 1 nest for every service remains in each region after the split. Each service has a redundancy level as shown in Table 2. The simulation of the logic's behavior is conducted in a custom built discrete event simulator.

Table 2. Service instances in the example

Service	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
# replicas	2	3	4	5	6	7	8	9	10	11

Mapping of the services is conducted in a network of 11 interconnected hosts, where we assume full mesh connectivity and do not consider the underlying network layer. The 11 nodes are partitioned into 5 domains as depicted in Fig. 2.

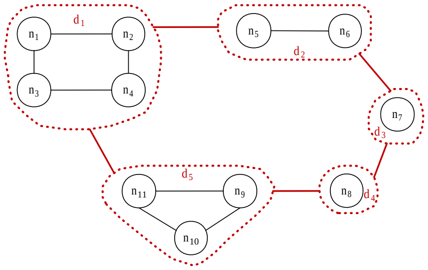
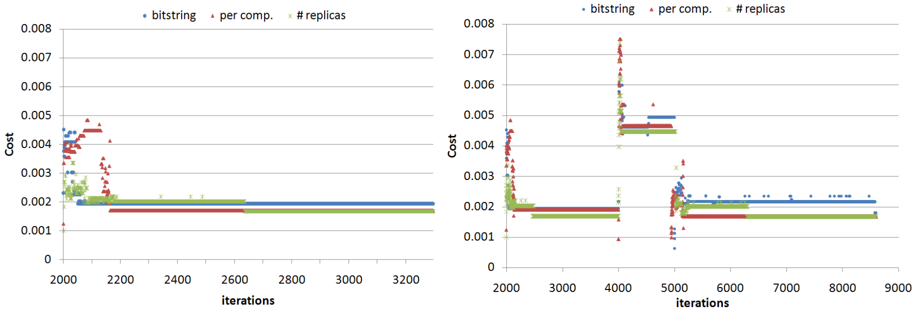


Fig. 2. Test network of hosts clustered into 5 domains

In this setting we conducted simulations with all three pheromone encodings. To test our concept of tackling domain splitting we have used a basic setting where domain d_1 containing 4 nodes has been split from the rest of the domains and later the two regions merged again. We then compared the resulting deployment mappings with the mappings obtained by executing our logic with no splitting. To demonstrate how the cost evaluation works in the optimization process the evolution of the cost output is displayed in case of service S_{10} in Fig. 3 with the three pheromone encodings introduced in Sec. 4.2. Fig. 3(a) shows how the optimal mappings are found and kept maintained iteration by iteration. The experiment is repeated with the introduction of the splitting of d_1 after 4000 iterations, the evolution of mapping costs is shown in Fig. 3(b).

An appropriate solution is found almost identically with the three different encodings. However, the $bitstring$ encoding converges to a solution with slightly higher overall cost, whereas the lowest cost is obtained first by $per\ comp.$ and somewhat later by $\#\ replicas$ too. In Fig. 3 the first 2000 iterations are not shown as the simulations start with 2000 explorer iterations for the sake of comparability. Initially, a random cost figure appears corresponding to exploration that is omitted here. The amount of initial exploration was constrained by the $bitstring$ encoding. The more compact encodings would require significantly less iterations, e.g. one tenth of that. In Fig. 3(b), where a domain splits at iteration 4000 we can observe how the swarm adapts the mappings to a more expensive configuration after the event has happened. Similarly, as the domains merge the deployment mappings are adapted to utilize a more optimal configuration. The $bitstring$ encoding in this test case is unable to find exactly the same mapping and converges to a somewhat more costly solution. $per\ comp.$ is the fastest to obtain the lowest cost mapping followed by the third encoding about 1000 iterations later.



(a) Without split (b) With split

Fig. 3. Mapping costs of S_{10}

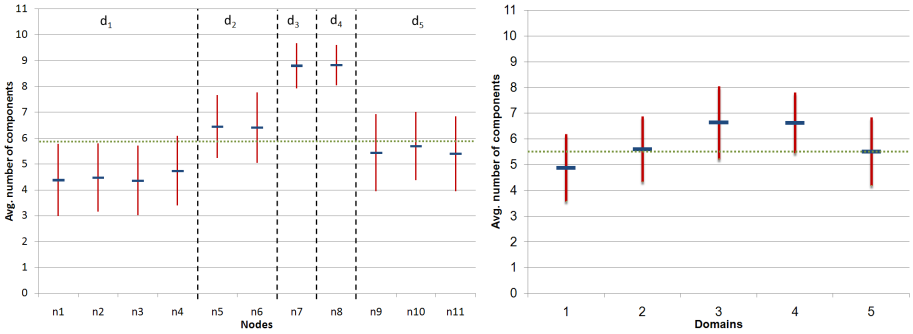
Considering the rules that we formulated regarding the dependability of the deployment mapping (cf. Sec. 2) Table 3 shows the three different pheromone encodings and the percentage of test cases, which succeeded in satisfying the two rules. The results are obtained by executing the algorithm 100 times with different input seeds.

Table 3. Success rate of the three encodings

wo/ split	ϕ_1	ϕ_2	w/ split	ϕ_1	ϕ_2
bitstring	100%	88%	bitstring	100%	87%
per comp.	100%	100%	per comp.	100%	100%
# replicas	100%	100%	# replicas	100%	99%

Our first objective was load-balancing among the nodes participating in the execution of the services, while basic dependability rules are satisfied too. To investigate that aspect we can look at the average number of replicas placed onto the nodes after convergence. Here, we chose the best encoding (cf. Table 3), i.e. $per\ comp.$. In Fig. 4(a) the average load placed on the 11 nodes $(n_1 \ldots n_{11})$ partitioned into the 5 domains is depicted. A total of 65 replicas constituted the ten service instances giving an average of 5.91 replicas per node; shown as a dotted horizontal line. We observed that the smaller domains, e.g. d_3, d_4, were overloaded compared to the rest due to ϕ_1, but generally replicas were placed quite evenly, showing that cooperation between the species worked.

As a next step towards developing our logic further for larger scales we repeated our experiment with a setting consisting of 50 nodes in 5 domains (containing 20-10-5-5-10 nodes respectively). Naturally, an increased amount of available resources for placement would make the deployment mapping problem actually easier, so we have used larger service specifications too to scale up the problem. Accordingly, the 10 services assigned to ant species were sized as $|C_i^k| = i \cdot 5$ replicas for S_i, where $i = 1 \ldots 10$, thus giving a total amount of 275 replicas.



(a) Over 11 nodes

(b) Over 5 domains (50 nodes)

Fig. 4. Load-balancing (average number of replicas and deviation per node)

We repeated the experiment 50 times using the selected encoding, $per\ comp.$, and allowing a maximum amount of 10000 iterations in each run. The resulting average execution load in the 5 available domains is depicted in Fig. 4(b), where the average number of replicas per node, using identical domains, would be 5.5 that is shown with a dotted horizontal line. Regarding load-balancing similar effects are observed as in the previous example. We can look at the dependability aspects of the solutions obtained in the 50 runs of the simulation too. As the problem size was significantly larger and the number of allowed iterations was constrained too, collocation is observed in some cases (in Table 4), while rule ϕ_1 is never violated. Violations of ϕ_2 are more frequent in case the number of replicas was close to the number of available nodes (e.g. S_{10}), which makes satisfying ϕ_2 harder when load-balancing has to be performed simultaneously.

Table 4. Collocation within the 10 large services

service	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}
collocation (ϕ_2)	0	1	0	3	1	1	0	3	1	13

6 Closing Remarks

Our focus has been on applying swarm intelligence, in particular the CEAS method to manage the deployment of collaborating software components. While developing our distributed approach we targeted a logic that shall not be over-engineered and uses only a few parameters that do not depend on the problem at hand (e.g. to avoid having to adjust parameters and cost functions manually). It is also required to be able to handle certain degrees of dynamics and adaptation to changes in the execution context. These are the reasons that lead us to nature inspired methods and systems that do not have to be altered significantly for every new target system. We have tested the ability of the logic to handle domain splitting and dealing with dependability requirements as well as load-balancing using two example settings and a custom built simulator. We believe that applying CEAS will not only result in a tailored optimization method but, at least on the long run, it will allow the implementation of a prototype of a truly distributed system that will support run-time deployment within software architectures.

Our results are promising and are inline with our efforts to further develop the deployment logic and increase its scalability and adaptability. Furthermore, we plan to experiment with another dimension of dynamicity by introducing run-time component replication that means that the amount of replicas in a service might change at run-time. Moreover, extensive simulations will be conducted to test scalability and convergence of the algorithm and also to evaluate its behavior compared to other relevant optimization methods that support distributed execution. This is a possible direction for future work, however, we advocate that a thorough comparison could in fact be a separate paper in itself as it would require fine tuning of multiple parameters in case of many available methods to be able to look into the scenario at hand with confidence.

References

1. Amazon Elastic Compute Cloud. Last checked: Sep 28, 2009. http://aws.amazon.com/ec2
2. J. Dean. Software engineering advice from building large-scale distributed systems. Last checked: Sep 28, 2009. http://research.google.com/people/jeff/stanford-295-talk.pdf
3. D. Fernandez-Baca. Allocating modules to processors in a distributed system. IEEE Tran. on Software Engineering, vol. 15, no. 11, 1989.
4. 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 (DAIS)*. LNCS 5523, Jun 2009.
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)*, ICST/ACM. Sep 2008.
6. P. E. Heegaard, B. E. Helvik and O. J. Wittner. The Cross Entropy Ant System for Network Path Management. *Telektronikk*, 104(01), pp. 19-40, 2008.
7. H. Meling and J. L. Gilje. A Distributed Approach to Autonomous Fault Treatment in Spread. In *7th European Dependable Computing Conference*, IEEE CS, May 2008.
8. R. Kusber, S. Haseloff and K. David. An Approach to Autonomic Deployment Decision Making. In *3rd Int'l Workshop on Self-Organizing Systems*, LNCS 5343, December 2008.
9. K. Joshi, M. Hiltunen and G. Jung. Performance Aware Regeneration in Virtualized Multitier Applications. In *DSN'09 Workshop on Proactive Failure Avoidance, Recovery and Maintenance (PFARM)*, IEEE CS, Jun 2009.
10. M. Dorigo, V. Maniezzo and A. Colorni. The Ant System: Optimization by a colony of cooperating agents. IEEE Tran. on Systems, Man, and Cybernetics Part B: Cybernetics, vol. 26, no. 1, 1996.
11. M. J. Csorba, P. E. Heegaard and P. Herrmann. Component Deployment Using Parallel Ant-nests. To appear in Int'l Journal on Autonomous and Adaptive Communications Systems (IJAACS). ISSN (Online): 1754-8640. ISSN (Print): 1754-8632. InderScience publishers. 2009.
12. R. Y. Rubinstein. The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability*, 2:127-190, 1999.
13. B. E. Helvik and O. Wittner. Using the Cross Entropy Method to Guide/Govern Mobile Agent's Path Finding in Networks. In *3rd Int'l Workshop on Mobile Agents for Telecommunication Applications*. LNCS 2164, Aug 2001.