

Foraging for better deployment of replicated service components

Máté J. Csorba¹, Hein Meling², Poul E. Heegaard¹, and Peter Herrmann¹

¹ Department of Telematics,

Norwegian University of Science and Technology, N-7491 Trondheim, Norway
{Mate.Csorba, Poul.Heegaard, Peter.Herrmann}@item.ntnu.no

² Department of Electrical Engineering and Computer Science,

University of Stavanger, N-4036 Stavanger, Norway

hein.meling@uis.no

Abstract. Our work focuses on distributed software services and their requirements in terms of system performance and dependability. We target the problem of finding optimal deployment mappings involving multiple services, i.e. mapping service components in the software architecture to the underlying platforms for best possible execution. We capture important non-functional requirements of distributed services, regarding performance and dependability. These models are then used to construct appropriate cost functions that will guide our heuristic optimization method to provide better deployment mappings for service components. This paper mainly focuses on dependability. In particular, a logic enabling replication management and deployment for increased dependability is presented. To demonstrate the feasibility of our approach, we model a scenario with 15 services each with different redundancy levels deployed over a 10-node network. We show by simulation how the deployment logic proposed is capable to satisfy replica deployment requirements.

1 Introduction

Distributed applications and services are increasingly being hosted by infrastructure providers over virtualized architectures, enabling on-demand resource scaling, such as in the Amazon EC2 platform [1]. An important concern in such platforms is the problem of *finding optimal deployment mappings* involving multiple services spread across multiple sites. During service execution a plethora of parameters influence the optimal deployment mapping, and more so in a distributed environment where concurrent services influence each other as well. Furthermore, some applications have non-functional requirements related to dependability, such as fault tolerance and high availability. Upholding such requirements demands replication protocols to ensure consistency, but also adds additional complexity to the optimization problem. Ideally, the deployment mappings should minimize the resource consumption, yet provide enough resources to satisfy the dependability requirements of services.

This paper presents a novel modeling and optimization methodology for deployment of replicated service components. We model services in a platform independent manner using the SPACE [3] methodology. As previously shown by Fernandez-

Baca [4], the general module allocation problem is NP-complete except for certain communication configurations, thus heuristics are required to obtain solutions efficiently. Based on our service models, we apply an heuristic optimization method called the Cross-Entropy Ant System (CEAS) [5], which is able to take multiple parameters into account when making a decision on the deployment mapping. 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.

There are a number of reasons to develop replicated services, including fault tolerance, high availability and load balancing. This work focuses on fault tolerance and availability, and in this context, the objective is to improve the availability characteristics of the service by appropriate allocation of service replicas to nodes, such that the impact of replica failures and network failures is reduced. And at the same time minimizing the resource consumption.

Generally, to support replicated services the underlying architecture needs to provide *replication protocols* to ensure consistency between replicas, e.g., active or passive replication protocols [16,2]. Such protocols have different implicit communication and computation costs and can be taken into account in our model. In addition, a *replication management* infrastructure, e.g. [11,10,9], is necessary to support deployment of replicas and managing reconfigurations when failures occur. One example is the distributed autonomous replication management framework (DARM) [9]. DARM focuses on the deployment and operational aspects of the system, where the gain in terms of improved dependability is likely to be the greatest. DARM is equipped with mechanisms for localizing failures and system reconfiguration. Reconfiguration is handled without any human intervention, and according to application-specific dependability requirements. The benefits of DARM are twofold: (i) the cost of deploying and managing highly available applications can be significantly reduced, and (ii) its dependability characteristics can be improved as shown in [6]. The approach presented in this paper can be combined with frameworks such as DARM in order to improve the deployment mapping operation; such an implementation has been left as future work.

There are at least three cases where finding suitable deployment mappings are of significance to replication management: (i) initial deployment of replicas according to some policy; (ii) reconfiguration of deployed replicas that have failed or become unavailable due to a network partition according to some maintenance policy; (iii) migration of replicas to re-balance the system load. The deployment mapping policy used in this paper, is formulated as a cost function to the optimization problem, essentially stating that replicas should be placed on nodes and domains (sites) so as to improve the dependability of the service being deployed.

The paper is organized as follows. The next section presents how replicas are modeled in the SPACE modeling framework. Sec. 3 introduces CEAS and provides a description of the deployment algorithm. Subsequently, we formulate the optimization problem and present cost functions used to solve it. Simulation results using our logic are presented in Sec. 5. Finally, in Sec. 6 we conclude and touch upon future work.

2 Replica services in SPACE

To account for dependability requirements while deploying replicated service components, collaboration-oriented models can be used. To this end, the SPACE [3] methodology provides a modeling technique for automated engineering of distributed applications. In contrast to other UML-based methods, it enables the composition of system descriptions from collaboration-oriented sub-models that does not specify the behavior of a single physical component, but rather describes the sub-functionality encompassing various system entities. Such sub-models are typically easier to reuse than component-oriented building blocks, since different systems in a particular domain often have similar sub-functions, which can be coupled in various ways. Each sub-function can then easily be specified as a collaborative building block once, and thus the creation of a new system can be reduced to the design of a new combination of these pre-defined blocks.

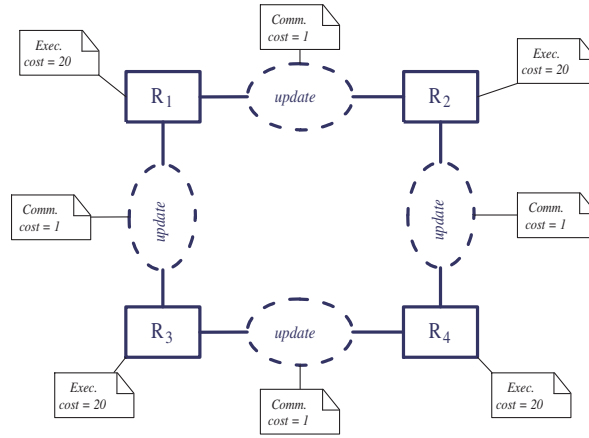


Fig. 1. Example service, S_1 with 4 replicas and corresponding costs

In SPACE, the topology of a system is modeled with UML collaborations, while behavior is described using UML activities. SPACE is accompanied by the Arctis [7] tool, which enables composition of models, various model checker-based correctness proofs and automated transformation to executable Java code.

In this paper, only UML collaborations are used. Fig. 1 depicts a simple example model. It describes the pair-wise replication of data between the physical components R_1 to R_4 . The updating function is modeled by four collaboration uses called *update*, each specifying the alignment of data between the two linked components. Although SPACE offers specification of multiple instances of a collaboration [8], for clarity only four instances of *update* are used here. SPACE models can be embellished with additional non-functional requirements that can be exploited by our deployment logic, i.e., the execution costs assigned to components or costs that are specific to each collaboration between replicas. Within a given service specification, some (or all) of the service components might require replication to improve their dependability. We propose to model and specify component replication using the same methodology applied for designing the services themselves. In other words, we specify a set of replicas related to a

specific component separately, i.e. as a collaboration of replicas of a single component.

To test our deployment logic, we assume an active replication approach, where each replica of the service performs according to the client requests. Thus, replicas have the same execution cost. Each replica is also assigned a communication (collaboration) cost to account for the cost of ensuring consistency (state updates) between replicas. The example scenario illustrated in Fig. 1 is used as a basis for the simulations presented in Sec. 5.

3 Replica Deployment using the Cross Entropy Ant System

To find suitable replica placements a collection of ant-like agents, denoted *ants*, search iteratively for the best solution according to a cost function, restricted by the problem constraints. To find a solution ants are guided using the analogy of *pheromones*, which are proportional to the quality of the solution. CEAS uses the *Cross Entropy method* for stochastic optimization introduced by Rubinstein [12], and has demonstrated its capabilities and relevance through a variety of studies of different path management strategies. For an intuitive explanation and introduction to CEAS, see [5].

Table 1 gives the notation for sets and variables used throughout our description.

Table 1. Notational shorthand

Shorthand	Usage	Description
\mathbf{S}	$S_k \in \mathbf{S}$	set of service instances
\mathbf{C}_k	$c_i \in \mathbf{C}_k$	set of all replicas in S_k
\mathbf{D}	$d \in \mathbf{D}$	set of all existing domains
\mathbf{N}	$n \in \mathbf{N}$	set of all existing nodes
$ \mathbf{C}_k $	$ \mathbf{C}_k $	number of replicas to be deployed
D_r	$d \in D_r$	list of domains used in deployment of S_k
N_r	$n \in N_r$	list of nodes used in deployment of S_k
NL_r	$nl_{n,r} \in NL_r$	load-level samples for S_k
M_r	$m_{n,r} \in M_r$	mapping list for S_k
H_r	$n \in H_r$	hop-list for S_k

In this paper, we apply CEAS to obtain the best mapping of a set of replicas onto a set of nodes, $M : \mathbf{C} \rightarrow \mathbf{N}$. The pheromone values used by the ants, denoted $\tau_{mn,r}$, correspond to a set of replicas, m mapped to node n at iteration r . Ants use a random proportional rule for selecting the individual mappings.

$$p_{mn,r} = \frac{\tau_{mn,r}}{\sum_{l \in M_{n,r}} \tau_{ln,r}} \quad (1)$$

The pheromone values $\tau_{mn,r}$ in (1) are updated continuously by the ants as follows:

$$\tau_{mn,r} = \sum_{k=1}^r I(l \in M_{n,r}) \beta^{\sum_{j=k+1}^r I(j \in M_k)} H(F(M_k), \gamma_r) \quad (2)$$

where $I(x) = 1$ if x is true, 0 otherwise. See [5] for further details.

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

$$H(F(M_r), \gamma_r) = e^{-F(M_r)/\gamma_r} \quad (3)$$

which is applied to all r samples.

To enable a distributed optimization process the cost of a mapping, $F(M_r)$ is calculated *immediately* after each sample i.e., when all replicas are mapped, and an auto-regressive performance function, $h_r(\gamma_r)$ is applied, Eq. 4.

$$h_r(\gamma_r) \approx \frac{1-\beta}{1-\beta^r} \sum_{i=1}^r \beta^{r-i} H(F(M_r), \gamma_r) \quad (4)$$

where $\beta \in (0, 1)$ is a *memory factor* weighting (geometrically) the output of the performance function. This mechanism smooths variations in the cost function, hence rapid changes in the deployment mapping and undesirable fluctuations can be avoided. The temperature, γ_r is determined by minimizing it subject to $h(\gamma) \geq \rho$, thus

$$\gamma_r = \left\{ \gamma \mid \frac{1-\beta}{1-\beta^r} \sum_{i=1}^r \beta^{r-i} H(F(M_i), \gamma) = \rho \right\} \quad (5)$$

where ρ is a parameter (denoted *search focus*) close to 0 (typically 0.05 or less).

Eq. (5) is a transcendental function that is storage and processing intensive since all observations up to the current sample, i.e., the entire mapping cost history $F(M_r), \forall r$ should be stored, and weights for all observations would have to be recalculated, thus putting an impractical burden on the on-line operation of the logic. Accordingly, we assume that, given a β close to 1, changes in γ_r are typically small from one iteration to the next, enabling a first order Taylor expansion of (5), and a second order Taylor expansion of (2), see [5] for more details. More importantly, we are able to obtain an optimal deployment mapping with high confidence, since CEAS can be considered as a subclass of Ant Colony Optimization (ACO) algorithms [13], 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 secured in a finite number of iterations.

Algorithm 1 Code for $Nest_k$

```

1: Initialization:
2:    $r \leftarrow 0$  {Number of iterations}
3:    $\gamma_r \leftarrow 0$  {Temperature}
4: while  $r < R$  {Stopping criteria}
5:    $antAlgo(r, \gamma_r)$  {Emit new ant}
6:    $r \leftarrow r + 1$ 

```

We now present the steps executed by the deployment logic to obtain a mapping of replicas. Behavior of the logic is separated into Algorithm 1, which describes the simple functionality of a *Nest*, i.e. basic additional intelligence in one of the nodes, and Algorithm 2, which describes the behavior of the ants that are subsequently emitted from the *Nest*. The role of a *Nest* can be played by an arbitrary node. The steps are executed independently by ants of each species, where a species is directly involved in the deployment of a specific service. Each ant initiated from the nest node of a species is assigned a set of replicas, \mathbf{C} ; in this case the replica instances to deploy. The ant then starts a random-walk in the network, selecting the next hop at random. Behavior at a node depends on if the ant is an *explorer* or a *normal* ant. *Normal* ants select a subset of \mathbf{C} for mapping to the current node according to the pheromone database and store this

selection in the mapping list, M_r . An *explorer* ant, however does the selection without using the pheromone values in a completely random manner.

Algorithm 2 Ant code for deployment mapping of component replicas $\mathbf{C} \in S_k \subset \mathbf{S}$ from $Nest_k$

```

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:    $NL_r \leftarrow \emptyset$                                {Set of load samples}

6: function antAlgo( $r, k$ )
7:    $\gamma_r \leftarrow Nest_k.getTemperature()$              {Read the current temperature}
8:   while  $\mathbf{C} \neq \emptyset$                                  {More replicas to deploy}
9:      $n \leftarrow selectNextNode()$                        {Select first node}
10:    if explorer ant
11:       $m_{n,r} \leftarrow random(\subseteq \mathbf{C})$            {Explorer ant; randomly select a set of replicas}
12:    else
13:       $m_{n,r} \leftarrow rndProp(\subseteq \mathbf{C})$              {Normal ant; select replicas according to Eq. (1)}
14:      if  $\{m_{n,r}\} \neq \emptyset, n \in d_k$                {At least one replica mapped to this domain}
15:         $D_r \leftarrow D_r \cup d_k$                        {Update the set of domains utilized}
16:         $M_r \leftarrow M_r \cup \{m_{n,r}\}$                {Update the ant's deployment mapping set}
17:         $\mathbf{C} \leftarrow \mathbf{C} - \{m_{n,r}\}$                  {Update the set of replicas to be deployed}
18:        if  $r \bmod 10 = 0$                                  {Only every 10th ant modifies allocations}
19:          foreach  $c_i \in m_{n,r}$ 
20:             $sumpp \leftarrow sumpp + f_{c_i}$                {Sum the exec. costs imposed by  $S_k$ }
21:             $n.reallocProcLoad(S_k, sumpp)$              {(re-)allocate processing power needed by  $S_k$ }
22:             $nl_{n,r} \leftarrow n.getEstProcLoad()$        {Get the estimated processing load at node  $n$ }
23:             $NL_r \leftarrow NL_r \cup \{nl_{n,r}\}$          {Add to the list of samples}

24:     $cost \leftarrow F(M_r, D_r, NL_r)$                    {Parameters depending on the cost function}
25:     $\gamma_r \leftarrow updateTemp(cost)$                  {Given cost, recalculate temperature according to Eq. (5)}
26:    foreach  $n \in H_r.reverse()$                          {Backtrack along the hop-list}
27:       $n.updatePheromone(m_{n,r}, \gamma_r)$              {Update pheromone value at  $n$ , Eq. (2)}
28:     $Nest_k.setTemperature(\gamma_r)$                    {Update  $\gamma_r$  at  $Nest_k$ }

29: function selectNextNode()                             {SELECT UNIQUE RANDOM NODE}
30:    $\mathbf{R} \leftarrow \mathbf{N} - currentNode$                    {Set of candidate nodes for ant traversal}
31:    $n \leftarrow random(\mathbf{R})$                              {Select candidate node at random}
32:    $H_r \leftarrow H_r \cup \{n\}$                          {Add node to the hop-list}
33:   return  $n$ 

```

The benefits of applying *explorer* ants are twofold, first they initially explore the solution space and second, they are used for faster discovery of changes in the network during optimization. In both cases, *explorers* do not use the pheromone tables, instead they build up an initial database. Besides, they are used to detect alternative solutions while the system undergoes short- or long-term changes. The amount of *explorer* vs. *normal* ants is a configurable ratio parameter to the logic. Initial exploration is essentially a random sampling of the problem space and the number of iterations depends on the problem size. However, the end of this phase can be detected by monitoring the

pheromone database size. Optimizing the deployment mappings based on the available cost functions should be performed using a distributed method, avoiding a centralized structure. To do so, each node provides a processing power reservation mechanism. Ant species use this mechanism to indicate their resource usage in every node they utilize for their replicas. Processing power reservation can be updated by a given percentage of ants, which is again a parameter to the logic, i.e., only a certain fraction (e.g. 10%) of iterations result in re-allocation at the nodes, see Lines 18 – 21 in Algorithm 2. Outdated allocations get invalidated in the nodes to preserve consistency. In addition, the allocation mechanism can serve as a means of interaction between the species. Thus, the current sum of allocations in a node can be sampled providing a general overview for the ants. These load-level samples are denoted NL_r . The decreased ratio of reservations by the ants (e.g. only 10% of them) contributes to obtaining a smoother series of NL_r samples. The actual implementation of sampling is left to the middleware.

The *forward search* phase of an ant is over when all component replicas are mapped and the resulting mapping is stored in M_r . The algorithm proceeds with evaluating the resulting mapping using the appropriate cost function $F_i()$. After evaluating the cost of the mapping, the ant *backtracks* to its nest using the hop-list, H_r . During *backtracking*, pheromone values distributed across the network of nodes are updated according to Eq. (2). After the ant finds its way back to the nest node or times out a new ant can be initiated and emitted. The same behavior can be used for all ants, even though they are of different species.

The main purpose of the pheromone database is its usage in Algorithm 2, Line 13. In every iteration, an ant will form $|N_r|$ discrete subsets of \mathbf{C} as it visits $n \subseteq N_r$ nodes. In order to be able to describe replica mappings to nodes, values of the pheromone database have to be aligned with replica sets. Accordingly, the pheromone database is built by assigning a flag to every replica available for deployment in a service, $\forall c_i \in \mathbf{C}$, with the exception of replicas that are bound to specific nodes explicitly by requirements and thus, they cannot be moved.

The pheromone database will contain $2^{|\mathbf{C}|}$ elements, equal to the number of possible combinations for a set c_i at a node, which is specific for each service. This determines a physical requirement for the execution platform that supports our logic, namely to be able to accommodate $2^{|\mathbf{C}|}$ floating point numbers for each of the services in every node. If the pheromone database in a node is normalized between $\{0 \dots 1\}$ it can be observed as a probability distribution of replica sets mapped to that node. In a converged state the optimal solution(s) will emerge with probability one.

4 Construction of the Cost Function

When applying the optimization method presented in Sec. 3 it is essential to formulate a proper cost function aimed at guiding the optimization process towards an *appropriate solution*. An *appropriate solution* is a solution to the deployment mapping problem satisfying the system requirements, F_{req} derived from the service specification, while accounting for the costs of the mapping, $F_i()$. Trying to find a global optimal solution does not make much sense in the systems considered here, as the solution would most likely be suboptimal by the time, the optimal mapping could be applied. However, the

algorithm can continue optimization even after a feasible mapping is found, that can trigger (re-)deployment of replicas. By optimal mapping we mean mappings with the lowest possible cost, while for a feasible mapping $F_i() < F_{req}$ is enough. Note that the formulation of the deployment problem below is independent of the methods we apply to obtain a solution.

$$\begin{array}{l} \min F_i() \quad \{ < F_{req} \} \\ \text{subject to } \Phi \end{array}$$

In each iteration of our deployment logic, the cost function is evaluated for every suggested mapping, $m_{n,r}$, (cf. Algorithm 2, Line 24). Properties of this function impact the quality of the solutions obtained as well as the convergence time, or in other words, the number of iterations required to reach a stable solution. In order to develop a logic that can aid replica deployment and increase dependability by influencing the mapping of software architecture the cost function has to be carefully selected. However, what is the proper function to use depends on the requirements and goals of the service. Here, we target efficient placement of component replicas in an active replication scheme aimed at improving the dependability.

We define the mapping functions f_k and $g_{k,d}$ as follows.

Definition 1. Let $f_k: r_k \rightarrow d$ be the mapping of replica r_k to domain $d \in \mathbf{D}$

Definition 2. Let $g_{k,d}: r_k \rightarrow n_d$ be the mapping of replica r_k to node $n_d \in \mathbf{N}$ in domain $d \in \mathbf{D}$

We then define two distinct rules that the deployment logic targets. The first one states that replicas shall be distributed across as many domains as possible for increased dependability, i.e. two replicas of the same service shall not be placed in the same domain preferably, or if there are more replicas than domains available there shall be at least one replica in all domains (ϕ_1).

Rule 1 $\phi_1: f_k \neq f_l \iff k \neq l \wedge |S_k| < |\mathbf{D}|$

Whereas the other rule declares that two replicas of the same service should not be co-located on the same node (ϕ_2).

Rule 2 $\phi_2: g_{k,d} \neq g_{l,d} \iff k \neq l, \forall d$

Deployment mappings of component replicas can be evaluated by the deployment cost function $F_i()$. Accordingly, we formulate the replica deployment problem as the task of minimizing $F_i()$ subject to $\Phi = \phi_1 \wedge \phi_2$.

The problem of producing deployment mappings that conform to the rules introduced above is approached step-wise by introducing different types of cost functions. We start by considering ϕ_1 only and use information collected by the ant species during *forward search* by counting the number of domains that have been used to map replicas at an iteration, this variable will be denoted D_r . Using D_r we will experiment with a reciprocal (6) and a linear function (7) too. The latter case uses the number of replicas, $|\mathbf{C}|$, a constant derived from the service model and thus known to each species.

$$F_1(D_r) = \frac{1}{|D_r|} \tag{6}$$

$$F_2(D_r, \mathbf{C}) = |\mathbf{C}| - |D_r| + 1 \quad (7)$$

Similarly, we include ϕ_2 into the cost function by a reciprocal and a linear function and combine it with (6) and (7) as follows.

$$F_3(D_r, N_r) = \frac{1}{|D_r|} \cdot \frac{1}{|N_r|} \quad (8)$$

$$F_4(D_r, N_r, \mathbf{C}) = (|\mathbf{C}| - |D_r| + 1) \cdot (|\mathbf{C}| - |N_r| + 1) \quad (9)$$

$$F_5(D_r, N_r, \mathbf{C}) = \frac{1}{|D_r|} \cdot (|\mathbf{C}| - |N_r| + 1) \quad (10)$$

$$F_6(D_r, N_r, \mathbf{C}) = (|\mathbf{C}| - |D_r| + 1) \cdot \frac{1}{|N_r|} \quad (11)$$

In (8)-(11) we utilize a variable, N_r , which denotes the number of nodes that have been used by a specific species for deploying replicas at iteration r , this is also reported by each ant during the *forward search* phase. We evaluate all four possible combinations of the reciprocal and linear functions targeting ϕ_1 and ϕ_2 .

The last combination of cost functions, in (12), is a combination of the simple reciprocal function in (6) targeting ϕ_1 combined with a more complex function used successfully in service component deployment [14].

$$F_7(D_r, M_r, NL_r) = \frac{1}{|D_r|} \cdot F_{lb}(M_r, NL_r) \quad (12)$$

F_{lb} uses two parameters that are updated in every iteration, the replica mapping set M_r and the load-level samples taken in the nodes visited by the ant ($n_j \in H_r$), denoted NL_r . This function accounts for the execution and communication costs derived from the service specification as introduced in Sec. 2. Correspondingly, the function consists of two main parts, node (NC) and collaboration related costs (LC).

$$F_{lb}(M_r, NL_r) = \left[\sum_{\forall n_j \in H_r} NC(n_j) \right] \cdot (1 + x \cdot LC) \quad (13)$$

where x is a parameter used to balance the effect of the LC term, as needed. The component LC is strictly local to each species and incorporates the collaboration costs

$$LC(M_r) = \sum_{j=1}^K I_j f_{k_j}; \text{ where } I_j = \begin{cases} 1, & \text{if } Collab_k \text{ external} \\ 0, & \text{if } Collab_k \text{ internal to a node} \end{cases} \quad (14)$$

Thus, the term LC will take into account communication costs (f_{k_j}) assigned to those collaborations that happen between different nodes only, in other words aiming at minimizing remote communication.

Costs related to execution of replicas, i.e., node local costs are incorporated into the first term in (13). Node local costs aim at achieving load-balancing among the nodes hosting replicas. Importantly, in this term only the subset of nodes an ant has actually visited (H_r) is taken into account, not the total amount of nodes. The term that is calculated individually for each of the nodes in H_r is shown in (15).

$$NC_{n_j}(NL_{n,r}) = \left[\sum_{i=0}^{NL_{n,r}(n_j)} \frac{1}{\sum_{\forall n_j \in H_r} NL_{n,r} + 1 - i} \right]^y \quad (15)$$

The term NC counteracts the other term in (13), LC , which puts weight on replica mappings that have as much as possible of the collaborations within the same node(s). NC has an effect of distributing replicas, thus equalizing execution load among the available nodes to the highest extent possible. This way two counteracting requirement types are tackled in the same function. The exponent y in (15) can shift the focus towards load-balancing against minimization of remote communication in collaborations. In the experiments in this paper we use $x = 10^{-5}$ and $y = 2$, which are adjusted to the cost values derived from the models, e.g. see the example service in Fig. 1.

Using F_{lb} we are able to smoothen the output of the cost evaluation executed for each iteration of the deployment logic. Its purpose is to ease convergence of the logic by making the solution space more fine grained, i.e. simplifying differentiation between very similar deployment mappings with nearly the same cost value.

The next section presents simulation results evaluating all the cost functions presented here using an example setup.

5 Simulation Results

To evaluate our approach and the proposed cost functions, we developed a test scenario. The scenario consists of a network of 10 identical nodes clustered into 5 domains (cf. Fig. 2). The 5 domains have 3, 2, 1, 1, 3 nodes. Using this network of nodes, each ant species executing Algorithm 2 is assigned a replica service for deployment. A set of 15 actively replicated services with redundancy levels shown in Table 2 is used for the evaluation.

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}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}
#replicas	4	6	4	4	4	5	5	6	6	6	6	7	8	9	10

For example, see S_1 in Fig. 1. Each replica within a service has identical execution cost, and all replicas have the same cost in all services. Similarly, the same is true for the communication costs, i.e. $f_{c_i} = 20, \forall i$ and $f_{k_j} = 1, \forall j$.

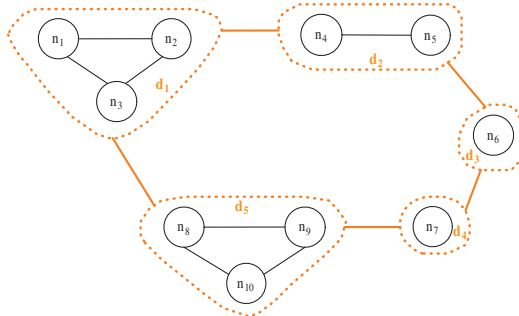


Fig. 2. Test network of hosts clustered into 5 domains

For the evaluation scenario with $S_1 \dots S_{15}$, the deployment logic (Algorithm 2) is executed 50 times using the cost functions discussed in Sec. 4 and we compare their behavior. The deployment logic was described by a process-oriented simulation model implemented in Simula/DEMOS [15].

For the problem at hand, deploying replicas of each service yield \mathbf{N}^{C_k} mapping combinations; deploying all 15 services simultaneously would account for an exhaustive search of $\mathbf{N}^{\sum C_k} = 10^{90}$ possible configurations. For the evaluation, the execution of Algorithm 2 was limited to $r_{max} = 30000$ iterations (significantly smaller than exhaustive search), unless convergence is obtained earlier. All 15 species, one for each service, were executed simultaneously. This is in accordance with our goal to find an appropriate solution within reasonable time, even though it may not be the optimal mapping. After each run, the obtained deployment mapping was checked against ϕ_1 and ϕ_2 . Results for selected functions are presented in Table 3.

Table 3. Replication rules satisfied, 50 trials each

Cost function	ϕ_1	ϕ_2	Comments
$F_1(D_r)$	88%	n/a	all due to no convergence
$F_5(D_r, N_r, \mathbf{C})$	100%	96%	all due to no convergence
$F_7(D_r, M_r, NL_r)$	100%	98%	all converged

In case of $F_1(D_r)$, which is based on observing the number of domains (D_r) utilized for deployment mapping of replicas, ϕ_2 (cf. Sec. 4) is not checked because the cost function does not consider this rule. From the 50 independent runs we see that in some cases ϕ_1 is not satisfied; some of the 15 services fail to utilize as many domains as they could. That is due to the limited number of iterations we allowed for the species to achieve convergence and because this cost function is very simple, i.e. lacking a more smooth, more fine grained evaluation of the deployment mappings for the ant species.

In the second branch of cost functions, Eq. (8)–(11), we apply two very simple functions together to take into account ϕ_1 and ϕ_2 at the same time. The experiments show that the combination of two functions of the same kind, i.e. two linear or two reciprocal functions, gives inferior results to applying a combination of one reciprocal and a linear. This might be caused by smoother cost output in case of the latter, which results in better convergence and better solution quality, i.e. a deployment mapping that satisfies the requirements with a higher probability. Nevertheless, there were 2 violations of ϕ_2 within the 50 runs, that means that one replica was co-located with another in one of the services. This is possible for services that have a high number of replicas, e.g. 9 or 10, which easily occupy 5 domains, thus obtain the lowest cost possible considering the first part of the cost function resulting in a mapping that violates ϕ_2 after convergence. These services, with these simple cost functions are able to decrease their mapping costs only marginally by spreading their replicas further among the available hosts, which results in sub-optimal solutions, thus violations of ϕ_1 or ϕ_2 .

Now, if we look at the last combination of functions in Table 3, we can see how our load-balancing function performed with the extension of taking into account the number of domains utilized (D_r). From the 50 independent runs the deployment logic converged to a stable solution in all of the cases. ϕ_1 was successfully taken into account by the first reciprocal term and resulted in no violations. In one case however, one of the services failed to satisfy ϕ_2 , i.e. a replica was co-located with another one. After a

closer look we can see that this involved service S_{15} comprising 10 replicas. The reason for this violation is that the load-balancing function, $F_{lb}(M_r, NL_r)$, has enforced a deployment mapping, which was better for global load-balancing in this particular case by taking into account this global goal to a greater extent and thus, violating the rule prohibiting co-location of replicas. However, as in S_{15} the number of replicas is equal to the number of available nodes there is not much space left for the logic to place replicas so that load-balancing is also achieved, which is the main goal for this part of the cost function. Clearly, applying the cost function we propose implies taking a broader view on the deployment problem. The tradeoff might be that under certain circumstances the mapping of replicas might violate one of the rules formulated, but the gain is that we can obtain a globally better and more effective mapping, still using a fully distributed logic and doing so faster, i.e. within reasonable time.

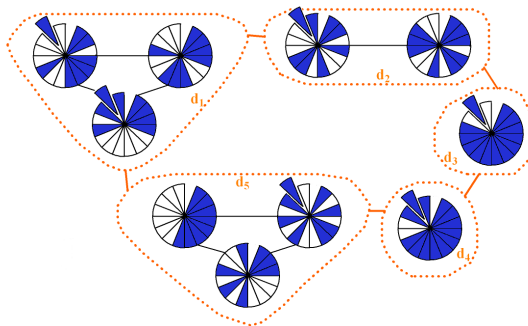


Fig. 3. Example mapping of replicas in $S_1 \dots SS_{15}$, with S_2 exploded

To get a picture of how replicas are mapped to the underlying nodes clustered into domains one of the possible mappings is depicted in Fig. 3, in which each slice of the pie diagram corresponds to a specific service $S_k \subset \mathbf{S}$. As in this optimal mapping there is no co-location of replicas, a slice being shaded means that there is a single replica placed on the particular node. It is easy to notice that the two domains consisting of a single node (d_3, d_4) are heavily packed with replicas due to the fact that there are many services, which can exploit 5 domains or more. This makes overall load-balancing among the available nodes more difficult.

Furthermore, to illustrate the behavior and convergence of our logic, in Fig. 4 we look at the cost output of some species that guides the mapping of replicas as a function of number of iterations.

The three services presented have 6 (S_{11}), 8 (S_{13}) and 10 (S_{15}) replicas to deploy. The first 2000 iterations, i.e. the *exploration phase* is not shown in the figure. After 2000 initial iterations optimization continues and the cost values decrease, thus indicating increasingly improved mapping of replica components. We stop the simulation where the costs do not improve anymore, in this particular case after approximately 10000 additional iterations. By checking ϕ_1 and ϕ_2 we can see that the mapping obtained in this run satisfies both. In case where the number of replicas is high, e.g. 10, values do not deteriorate too much from a consensus level between the parallel species (considering $N = 10$) as ϕ_1 restricts the solution space. That means that for service S_{15} all of the nodes have to host one replica according to the rule. Whereas services with less replicas to deploy have a significantly larger valid solution space, i.e. service S_{11}

and S_{13} find, in some cases, a solution satisfying ϕ_1 and ϕ_2 but with a higher overall cost, thus we can see some deviations in the cost output in the figure before obtaining convergence. After consensus is reached among the species the actual deployment of component replicas can be triggered. Practically, when a species detects that the cost values obtained by its ants are stable over a period of time, replicas corresponding to this species can be (re-)deployed. The technical solution as well as the protocol of replica placement/(re-)deployment is, however, left to the middleware (e.g., DARM).

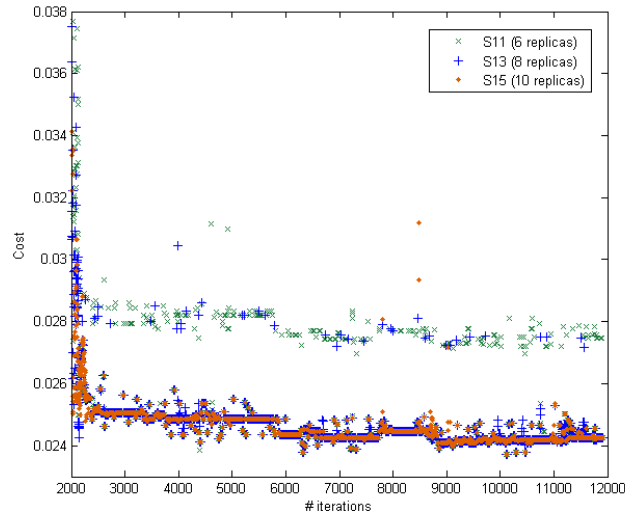


Fig. 4. Costs with 6, 8 and 10 replicas

6 Closing Remarks

Our focus has been on a heuristic optimization technique aided by swarm intelligence that can manage deployment of software components, in particular component replicas for increased dependability. To obtain an efficient mapping of replicas we utilize service models specified as UML 2.0 collaborations. These models are enriched with non-functional requirements that are used in the cost evaluation of the mappings made by the deployment logic. Importantly, our method is a fully distributed approach, thus it is free of discrepancies most of the existing centralized solutions suffer from, e.g. performance bottlenecks and single points of failure. Instead of having a centralized database we use the analogy of pheromones used by foraging ants as a distributed database across the network of hosts. This database can be quite compact as all the intelligence is carried along by the ant-like agents.

We have showed that, using CEAS for optimization and SPACE for modeling, the deployment logic is capable of handling various non-functional requirements present in service specifications. Extending on our previous work, in this paper focus has been on how the logic can deal with basic dependability requirements concerning replication management. Eventually, our goal is to aid run-time (re-)deployment and replication of software components while considering the execution environment and satisfying the

requirements of the service.

For future work we plan to introduce new types of species corresponding to user demands towards the services being deployed. However, this will introduce new challenges as it will increase the dimensions of the deployment problem significantly. More fine grained cost modeling, e.g. passive replication, with costs dependent on replica attributes will be part of future investigations. This will also involve more extensive simulations with new experimental settings. Larger network sizes will also be investigated together with their impact on convergence and scalability. Another interesting aspect we will experiment with is how splitting/merging of domains influences the output of our logic, besides assessing what level of node churn can be tolerated by our method. Generally, context-aware adaptation is considered one of the main tracks we follow in our future work.

References

1. Amazon Elastic Compute Cloud. <http://aws.amazon.com/ecs2>.
2. N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, ch. 8, pp. 199–216. Addison-Wesley, 2nd ed., 1994.
3. F. A. Kraemer, P. Herrmann. Service Specification by Composition of Collaborations - An Example. In Proc. IEEE/WIC/ACM Int'l Conference on Web Intelligence, Int'l Workshop on Service Composition (Sercomp'06), pp. 129-133, Hong Kong, IEEE CS, Dec. 2006.
4. D. Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, vol. 15, no. 11, 1989.
5. P. E. Heegaard and B. E. Helvik and O. J. Wittner. The Cross Entropy Ant System for Network Path Management. *Teletronikk*, 104(01), pp. 19-40, 2008.
6. B. E. Helvik, H. Meling, and A. Montresor. An Approach to Experimentally Obtain Service Dependability Characteristics of the Jgroup/ARM System. In Proc. *5th European Dependable Computing Conference*, vol. 3463 of LNCS, pp. 179–198. Springer-Verlag, Apr. 2005.
7. F. A. Kraemer and R. Bræk and P. Herrmann. Compositional Service Engineering with Arctis, to appear in *Teletronikk*, 2009
8. F. A. Kraemer, R. Bræk, P. Herrmann. Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications. In Proc. 13th System Design Language Forum 2007, pp. 166-185, LNCS 4745, Paris, September 2007.
9. H. Meling and J. L. Gilje. A Distributed Approach to Autonomous Fault Treatment in Spread. In Proc. *7th European Dependable Computing Conference*. IEEE CS, May 2008.
10. 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.
11. OMG. Fault Tolerant CORBA Specification. OMG Document ptc/00-04-04, Apr. 2000.
12. R. Y. Rubinstein. The Cross-Entropy Method for Combinatorial and Continuous Optimization. *Methodology and Computing in Applied Probability*, 1999.
13. M. Dorigo, et al. The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, vol. 26, no. 1, 1996.
14. M. J. Csorba and P. E. Heegaard and P. Herrmann. Adaptable model-based component deployment guided by artificial ants. In Proc. 2nd Int'l Conf. on Autonomic Computing and Communication Systems (Autonomics), ICST/ACM, Turin, September 2008.
15. G. Birtwistle. Demos - a system for discrete event modelling on simula. 1997.
16. F. B. Schneider. Replicated Management using the State-Machine Approach. In S. Mullender, editor, *Distributed Systems*, ch. 7, pp. 169–198. Addison-Wesley, 2nd ed., 1994.