New Generation Computing, 29(2011)185-222
Ohmsha, Ltd. and Springer

A Bio-inspired Method for Distributed Deployment of Services

Máté J. CSORBA
Department of Telematics,
Norwegian University of Science and Technology,
N-7491 Trondheim, NORWAY
Hein MELING
Department of Electrical Engineering and Computer Science,
University of Stavanger, N-4036 Stavanger, NORWAY
Poul E. HEEGAARD
Department of Telematics,
Norwegian University of Science and Technology,
N-7491 Trondheim, NORWAY
{Mate.Csorba, Poul.Heegaard}@item.ntnu.no,
Hein.Meling@uis.no

Received 5 October 2010
Revised manuscript received 8 January 2011

Abstract We look at the well-known problem of allocating software components to compute resources (nodes) in a network, given resource constraints on the infrastructure and the quality of service requirements of the components to be allocated to nodes. This problem has many twists and angles, and has been studied extensively in the literature. Solving it is particularly problematic when there is extensive dynamism and scale involved. Typically, heuristics are needed.

In this paper, we present a new breed of heuristics for solving this problem. The distinguishing feature of our approach is a decentralized optimization framework aimed at finding near optimal mappings within reasonable time and for large scale. Three different incarnations of the problem are explored through simulations. For one problem instance, we also provide exact solutions, and show that our technique is able to find near optimal solutions with low variance. In the largest example, a public-private cloud computing scenario is used, where different clouds are associated with financial costs, and we show that our approach is capable of balancing the load as expected for such a scenario.

Keywords: Service Deployment, Biologically-inspired Systems, Decentralized Optimization.

§1 Introduction

Many popular web applications and services are currently being deployed in large-scale data center environments due to the inherent scaling needs of such applications. Moreover, data center environments may consist of hundreds of thousands of nodes, and at this scale, there is bound to be a constant flux of topology changes due to scheduled maintenance and failures, and dynamism due to varying usage patterns and characteristics of the different applications deployed within the hosting data centers. Accounting for *all* the parameters involved in such a system is a challenging undertaking, and flexible methods are necessary to maintain the desired Quality of Service (QoS) levels at acceptable costs. A fundamental problem in this scenario is the *mapping of components*[*1] *to nodes* within the hosting data center(s), while accounting for the necessary tradeoffs that characterize the environment and the services to be deployed. This is what we refer to as *the deployment problem*.

The main contribution of this work is a bio-inspired, decentralized optimization technique for solving the deployment problem. The method is a search-based heuristic aimed at finding near optimal mappings even under harsh network conditions. We explore several incarnations of this problem through simulations; each at a different level of granularity and targeting different QoS requirements, as a mean to demonstrate the flexibility of our approach. For comparison, one instance of the problem is also solved with a traditional centralized optimization technique that finds the exact optimum. We show that for three problem sizes, the simulations of our decentralized approach can still achieve close to the optimal value, with low variance. Note however, techniques that find the exact optimum fall short as the problem size grows, whereas our heuristic is shown to scale conveniently to large problem sizes. Moreover, due to the inherent dynamism that exists in the systems we are targeting, a solution would quickly become suboptimal. However, our approach is able to rapidly adapt to changes in the environment, e.g. a network partition/merge event, by recalculating the deployment configurations, constrained by a threshold reflecting the migration costs. This ability of our approach is invaluable for deploying services in large-scale data center networks.

Our decentralized optimization framework is built around the Cross Entropy Ant System (CEAS),[15,18] which is derived from Ant Colony Optimization (ACO).[11] CEAS uses ant-like agents, denoted *ants*, that can move around in the network, identifying potential locations where components might be placed, and leave pheromone trails as a means to facilitate indirect communication about suitable locations for placement. The CEAS framework requires the definition

[*1] A component is assumed to be any software component that can be placed on a physical compute node, e.g. a building block of a service or application, or a virtual machine. Herein these concepts are used somewhat interchangeably.

of a cost function for the specific optimization problem at hand. The purpose of the cost function is to evaluate the utility of a given deployment configuration during the optimization process, eventually leading to the preferred deployment configuration. The proper selection of these functions is crucial for guiding the search for a deployment mapping that satisfies the QoS requirements of the service, which can then be deployed using some execution framework. Essential are also the constraints of the problem; be it load balancing or a certain availability level.

The remainder of the paper is organized as follows. In the following, we further elaborate on the deployment problem and place it in the context of a software development and deployment cycle. Then in Sec. 1.2, we discuss the complexity and scalability issues that are facing us in solving large scale deployment problems, followed by a discussion on how to capture and consider costs and requirements of the services that are deployed. In Sec. 1.4, we present candidate target environments for our deployment framework. Cost function design is discussed in Sec. 2, while Sec. 3 gives a general introduction to the CEAS framework that we build upon and presents a general definition of our algorithm. In Sec. 4, three example scenarios are shown: (i) deployment of collaborating components with communication costs (Sec. 4.1), (ii) deployment of replicas constrained by dependability requirements (Sec. 4.3), followed by (iii) our approach to mapping virtual machines in a public-private cloud computing environment (Sec. 4.4). In Sec. 4.5, we present a centralized approach for finding optimal mappings under certain sets of requirements and validate our results obtained by the decentralized algorithm. Finally, we review some related research and conclude.

1.1 The Deployment Problem

As described initially, we target the problem of mapping subsets of services to physical resources, i.e. nodes capable of hosting such services, in an efficient manner such that the QoS requirements of the services are satisfied. This problem can be viewed in the context of the software development and deployment cycle as partially illustrated in Fig. 1, where each layer captures a part of the multifaceted deployment problem. The execution nodes are shown in the bottom part, whereas the services to be deployed are modeled at the second layer up. The goal then is to obtain the mapping $M : \mathbf{C} \rightarrow \mathbf{N}$ between the building blocks of the service (the set $\mathbf{C}$) and the available nodes ($\mathbf{N}$).

The non-functional requirements of services are captured in the QoS dimensions layer; these can be issues such as dependability, security, performance, or energy-saving, all of which contribute to increasing the problem size and complexity. Finally, on the top layer we have the possible varieties of usage scenarios that can be captured by enriching the service models with additional usage related information, e.g. arrival rates for a component that handles user requests.

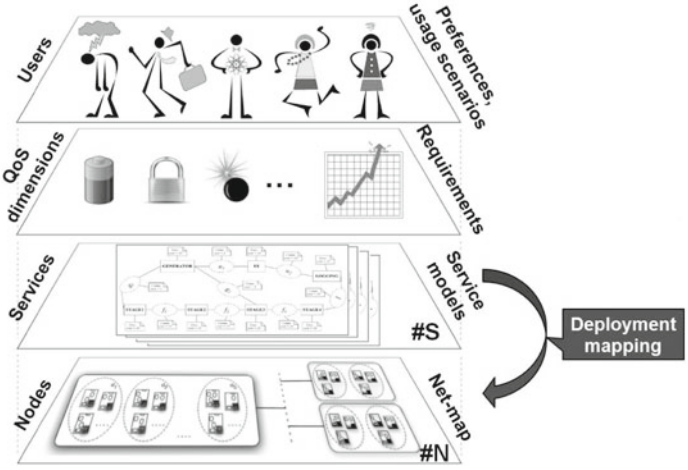


Fig. 1 The Multiple Dimensions of the Deployment Problem

1.2 Problem Complexity and Scalability Issues

The deployment mapping problem can be formulated as a multi-dimensional bin packing problem,[22] where each physical node is a bin and each constraint spans a dimension. The components with their associated resource requirements are the objects that are to be packed in the bins (nodes) available in the system. Thus, since bin packing is NP-hard,[22] the deployment mapping problem is also NP-hard. Even determining the existence of a valid packing is itself a NP-hard problem.[38] Moreover, the general module allocation problem has been shown to be NP-complete,[14] except for some communication configurations.

Given the complexity of solving the deployment mapping problem, it is to be expected that it cannot be solved even for moderately large scenarios. And especially not in realistic scenarios, such as finding efficient mappings in large-scale data center infrastructures, as the problem size grows exponentially with the number of hosting nodes and the number of components to be deployed. And also because a multitude of services are deployed simultaneously, while ensuring proper balance between load characteristics and service availability at every data center site.

Moreover, mappings found by an algorithm can be affected by a plethora of parameters during execution, e.g. due to the influence of concurrent services. Also, introducing dependability requirements for the services results in additional complexity, e.g. due to the use of replication protocols and their need to ensure consistency. Thus, given the problem complexity and for the relevant problem sizes, our only option is to look for efficient heuristic algorithms instead of seeking to find the exact optimum.

Furthermore, the heuristic algorithm should also be resilient to dynamics, or specifically, it should exhibit some degree of autonomy and adapt to changes in the environment. Such changes might include mobility of users, node churn, network disconnection (split/merge), incremental scaling of services, among oth-

ers. Thus, our approach to develop an autonomic process for tracking the best solutions in a dynamic environment is to build on the inherent self-organization properties of the CEAS framework. This framework facilitates decentralized optimization, avoiding the need for any centralized information storage and decision making. To cope with large scale problems however, an efficient data representation (storage) is necessary at each node participating in the decentralized algorithm. We discuss and evaluate three different data representations in Sec. 3.3.

1.3 Costs and Constraints

Given a specific deployment problem, our optimization technique, or deployment logic, must account for a range of parameters representing the QoS requirements and constraints of the services being deployed. In this section, we describe how these requirements and constraints are captured and give a few examples.

The QoS requirements are captured at design time and specified in a collaboration-oriented design model. The requirements may represent qualities such as security, performance, availability, portability, etc. In fact, our deployment logic can capture any kind of system property, as long as a suitable cost function can be defined for it. Figure 2 shows a sample collaboration diagram, enriched with two cost function attributes (QoS requirements), namely *execution* and *communication cost*. The execution cost captures the CPU requirement of a component to be deployed, whereas the communication cost reflects its network usage requirement.

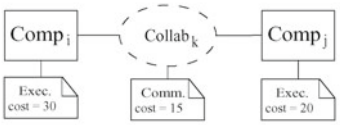


Fig. 2 A Collaboration Diagram with Non-functional Requirements

A service may constrain the placement of its components to specific nodes by means of a binding, e.g. a database server may have to be assigned to a specific node. Such a binding generally reduces the search space. However, bound components are still accounted for in the cost of a computed deployment mapping.

Our approach is designed as a continuous optimization process in order to facilitate rapid response to dynamism; hence reconfiguration of initial component placements are expected. However, to avoid migrating components for marginal cost savings, the cost of migration must be accounted for; herein we use a simple threshold scheme.

The three example scenarios presented in Sec. 4 use different requirements. The main overall aim is to load-balance execution cost across the available nodes, while accounting for all services running in the environment, remote communication costs, and dependability of the service being deployed. The second scenario introduces replica management rules to enforce cluster- and node disjointness.

In the last example, we extend the set of requirements to include financial costs of using different clusters for placement in a public-private cloud computing setting. Without loss of generality, all cost functions are formalized as minimization problems, i.e. the less the cost the better the solution.

1.4 System Model and Target Systems

This section describes the system model and gives some notation, followed by a brief description of potential target systems for which our deployment logic will be of relevance.

We consider the elementary building blocks (components) of a service as the unit to be deployed. Each component has well-defined interfaces and communicates by means of message exchange. A *service* is defined as the collaboration among its constituent components, which may be distributed across a network of nodes. This network of nodes offers an execution environment for the services, and can be organized into different network topologies, e.g. multiple clusters and/or clouds, depending on the usage scenario. Our optimization technique does not facilitate any means to find the optimal topology, but instead will find near optimal placements of components within a given topology. It will however, leverage knowledge about the topology in terms of constraints and requirements, e.g. specifying peak load scenarios and dependability requirements. Changing the execution context might dictate addition or removal of service instances.

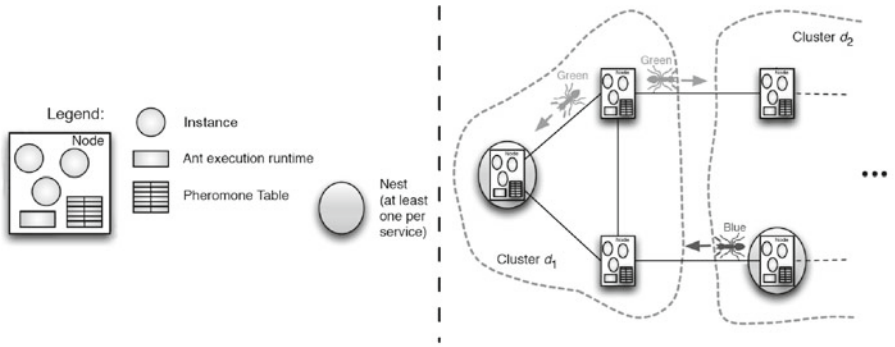


Fig. 3 Example Target Network

We model the system as a collection, $n_i \in \mathbf{N}$, of interconnected nodes. $\mathbf{N}$ is partitioned into a set $\mathbf{D}$ of *clusters*, as illustrated by d_1 and d_2 in Fig. 3. Clusters are usually formed according to geographical location or otherwise distinct administrative region. Let $\mathbf{S}$ denote the set of services to be deployed. The objective from Sec. 1.1 is thus to deploy a set of components $\mathbf{C}$ providing service $S_l \in \mathbf{S}$, and likewise for all services.

As shown in Fig. 3, every node has an *execution runtime* used to support installation, optimization and execution. Furthermore, for every service at least one instance of the CEAS, referred to as *species*, is run. CEAS's autonomous agents searching for a deployment of a given service are shown in different colors

in Fig. 3. For each of the species at least one designated node has to be present, serving as a home location for the agents, called the *nest*. A node also maintains information about the goodness of mappings in an information table (called the *pheromone table*, for details see Sec. 3.3), and capacity for installing one or more components of arbitrary services.

There exists a range of software execution frameworks and middleware that are potential candidates for integration with our deployment logic. For example, in case of collaborating software components one such candidate is the MUSIC middleware platform[31] that supports some self-* properties and component based software. When dependability aspects are of relevance, fault tolerant and self-repairing systems such as the DARM platform[28] can take advantage of our deployment logic. Finally, we mention a cloud computing scenario, where the deployment logic is used to optimize placement of VM instances. Candidate platforms include, e.g. Amazon EC2[2] and VMware. Nevertheless, the aim of the deployment logic is to be agnostic to the execution environment, i.e. optimization of the mappings can be done based on service models, regardless of the underlying platform. Example scenarios relevant to these execution environments will be presented in Sec. 4, followed by an evaluation based on simulations. Having introduced our targeted architectures, we now shift our attention to the construction of cost functions and how they are used throughout the optimization process. But first, we summarize our nomenclature in Table 1.

Table 1 Nomenclature

Shorthand	Usage	Size	Description
$\mathbf{S}$	$S_l \in \mathbf{S}$	$\lvert\mathbf{S}\rvert$	Set of services to deploy
$\mathbf{C}$	$c_i \in \mathbf{C}$	$\lvert\mathbf{C}\rvert$	Set of components in S_l
$\mathbf{K}$	$k_j \in \mathbf{K}$	$\lvert\mathbf{K}\rvert$	Set of collaborations in S_l
$\mathbf{N}$	$n \in \mathbf{N}$	$\lvert\mathbf{N}\rvert$	Set of all existing nodes
$\mathbf{D}$	$d \in \mathbf{D}$	$\lvert\mathbf{D}\rvert$	Set of all existing clusters
M	$m_{n,r} \in M_r$	$\lvert M\rvert{=}\lvert\mathbf{C}\rvert$	Mapping $\mathbf{C} \rightarrow \mathbf{N}$ in iteration r
D	$d \in D_r$	$\lvert D\rvert \leq \lvert\mathbf{D}\rvert$	List of clusters used in M
H	$n \in H_r$	$\lvert H\rvert \leq \lvert\mathbf{N}\rvert$	Hop-list of nodes visited
L	$l_{n,r} \in L_r$	$\lvert L\rvert = \lvert\mathbf{H}\rvert$	Load samples taken in iteration r

§2 The Cost Function

We now discuss the construction of cost functions for our optimization technique. A cost function, denoted $F()$, aims to evaluate the utility of a certain deployment configuration, and is used in each iteration of CEAS. Cost function design constitutes an important part of our work, and it is crucial to our technique to capture all aspects of the optimization problem. It has significant influence on the quality of the solutions and on the convergence rate. Recent work on autonomic computing also uses utility functions,[23] however, our decentralized approach demands more sophisticated functions due to lack of global knowledge at the decision logic. Within the application scenarios considered the cost values that describe a given deployment are to be minimized, i.e. the objective becomes $min\ F()$.

The cost function used by our technique is configured as a combination of several functions, typically one for each requirement/constraint dimension. Thus, the difficulty of efficient design is multifaceted. One is to find the appropriate granularity for each requirement/constraint dimension of a service. Furthermore, ensuring fast convergence dictates keeping the functions as simple as possible, yet the output of the functions must be fine grained enough to distinguish between two significantly different mappings. In some cases, an abundance of possible mappings exists with different but very similar qualities, which may lead to a non-linear cost function. Non-linearities can render their evaluation computationally expensive, causing a significant slow down in execution of the logic. Particular care has to be taken to weld components of the cost function together yielding a single function that is computationally effective and, at the same time, represents all the QoS requirements weighted by their importance. To achieve this, the most common combinations are multiplicative or additive combination of cost function components. We begin with a simple function that uses some global knowledge and considers deployment of a single service and then extend the function gradually.

2.1 Load-balancing and Communication Costs

In the first cost function, we consider only execution and communication costs, and we use this to build more complex functions. Assume the deployment logic has access to a service model specifying execution costs, e_i for each component $c_i \in \mathbf{C}$, and communication costs, f_j for each collaboration $k_j \in \mathbf{K}$. The total offered execution load for a given service is then $\sum_{i=1}^{|\mathbf{C}|} e_i$. Hence, the average load T in a network $\mathbf{N}$ becomes

$$T = \left\lfloor \frac{\sum_{i=1}^{|\mathbf{C}|} e_i}{|\mathbf{N}|} \right\rfloor \tag{1}$$

To quantify remote communication costs, we first introduce the indicator function $I(j)$, where $I(j) = 1$ if collaboration k_j is remote and $I(j) = 0$ if k_j is internal to a node. That is, we assume the cost of node internal communication is negligible ($I(j) = 0$), and thus, only consider the execution cost needed by these components, whereas for remote communication both costs are accounted for. To determine which collaboration k_j is remote the set of mappings, M is used. Given $I(j)$, the remote communication cost, $\Omega(M)$, covering all collaborations of the service is simply the sum

$$\Omega(M) = \sum_{j=1}^{|\mathbf{K}|} I(j) \cdot f_j \tag{2}$$

Thus, the combined cost function becomes

$$F_1(M) = \sum_{n=1}^{|\mathbf{N}|} |\hat{l}_n - T| + \Omega(M) \tag{3}$$

where $\hat{l}_n$, $n = 1 \ldots |\mathbf{N}|$ represent CPU load samples, and M is the mappings to evaluate. That is, load samples describe the execution load impact of the components mapped to a given node n, i.e. $\sum_i e_i$, for $\forall i$ where $c_i \rightarrow n$.

2.2 Multiple Services and Eliminating Global Knowledge

So far, we covered QoS requirements captured in the modeling phase of a single service. Next, we extend the function to multiple services and eliminate the need for global knowledge.

Finding T in (3) requires global knowledge of all the services deployed simultaneously. To eliminate the need for this global knowledge, we simply replace T with a set of load samples $l_n \in L$. To capture this notion of load sampling, we introduce a reservation mechanism in the optimization process. When components of a service are mapped to a node n_i, resources equivalent to the weights of the corresponding components will be reserved; the reservations are maintained in the actual nodes. CEAS can use this mechanism to estimate the resource usage on nodes, and to facilitate interaction between the different species. Agents in CEAS can then sample the current reservations on a node, and use this to evaluate the cost of a mapping involving that node. Samples that would exceed the capacity of a node are quickly outranked by better solutions as a high penalty is assigned to infeasible mappings. To keep the resource reservations current, we also add a timestamp-based eviction mechanism to prevent stale reservations.

To further improve scalability, only a portion of the network is considered, represented by the hop-list H, such that $|H| \leq |\mathbf{N}|$; the hop-list correspond to the set of nodes visited to obtain load samples. Thus, the upper bound of the first summation in (3) is reduced to $|H|$. In practice, this means that we are able to achieve near-optimal results without having to sample all $|\mathbf{N}|$ nodes in the network. This sampling scheme is applied to our example scenarios in Sec. 4.2-4.4.

To enable simultaneous deployment of multiple services, one species is run for each service. These species interoperate to obtain a more holistic view of their environment. To facilitate this cooperation, each species stores information at participating nodes. The objective of each species is to find a satisfactory mapping for the components of its service, while accounting for services being deployed concurrently.

Formulating a cost function partly depends on the parameters used; we use the deployment mappings in M and the load samples L. We reformulate the cost function as a multiplicative function instead of the additive version, presented in (3), as follows in (4). For more on experiments with additive and multiplicative functions we refer to [10].

$$F_2(M, H, L) = \Big[\sum_{\forall n \in H} C_0[n] \Big] \cdot (1 + \omega \cdot \Omega(M)) \tag{4}$$

where ω is a scaling parameter for $\Omega(M)$. $F_2(M, H, L)$ has a component, $\Omega(M)$

identical to (2), incorporating the communication costs individually for each service. Function $C_0[n]$, defined in (5), quantifies the node local costs for node n. Samples are obtained over a subset $H \subseteq \mathbf{N}$. This function targets load balancing among nodes.

$$C_0[n] = \Big(\sum_{i=0}^{L[n]} \frac{1}{\Theta_0 + 1 - i} \Big)^2 \tag{5}$$

where Θ_0 is the sum of load samples $\Theta_0 = \sum_{\forall n \in H} L[n]$. The first term in (4) counteracts the term for communication costs, $\Omega(M)$. The effects of these two counteracting terms can be balanced using the scaling parameter ω. The quadratic nature of C_0 allows shifting focus from minimizing the communication costs to load balancing.

2.3 Dependability Rules

We now turn our attention to dependability requirements; each component of a service may be replicated for fault tolerance and/or load-balancing. We also refine our view of the network by introducing *clusters* of nodes, $\mathbf{D}$. Each cluster may represent separate geographic sites. In this context, the aim of the deployment logic is to satisfy the dependability requirements specified in the service model and obtain an efficient mapping in the network, for an example see Sec. 4.3.

To support this scenario, the cost function must also accommodate the additional requirements. These requirements are specified as a set of dependability rules, denoted Φ, that constrain the minimization problem as $min\ F()$ subject to Φ. We define Φ with the aid of two mapping functions (that apply to a given service k).

DEFINITION 1 *Let $f_{i,d} : c_i \rightarrow d$ be the mapping of replica c_i to cluster $d \in \mathbf{D}$.*

DEFINITION 2 *Let $g_i : c_i \rightarrow n$ be the mapping of replica c_i to node $n \in \mathbf{N}$.*

The first rule, ϕ_1 requires replicas to be dispersed over as many clusters as possible, aimed at improving service availability despite potential network partitions. We assume that network partitions are more likely to occur between cluster boundaries. More specifically, replicas of a component should be placed in different clusters. If the replication degree exceeds the number of available clusters, at least one replica should be placed in each cluster. The second rule, ϕ_2, simply prohibits collocation of two replicas on the same node. Formally,

RULE 1 $\phi_1 : \forall d \in \mathbf{D}, \forall c_i \in \mathbf{C} : f_{i,d} \neq f_{u,d} \ \Leftrightarrow (i \neq u) \wedge |\mathbf{C}| < |\mathbf{D}|$

RULE 2 $\phi_2 : \forall c_i \in \mathbf{C} : g_i \neq g_u \ \Leftrightarrow (i \neq u)$

Let $\Phi = \phi_1 \wedge \phi_2$ be the set of dependability rules considered. Note that, prohibiting collocations by ϕ_2 is contradictory to minimizing remote communication. Thus, when considering dependability, we omit communication costs, e.g. in Sec. 4.3 and 4.4. To cater for these new rules, the number of utilized clusters,

referred to as $|D|$, is added as a parameter to the cost function. Several combinations of reciprocal and linear functions were evaluated in [10]. The function that gave the best results is a combination of a reciprocal term targeting ϕ_1 – using $|D|$, and the load-balancing function in (5), applied in two different ways. First, we redefine (5) applicable for each node n – previously used solely for load-balancing – to cater for ϕ_2 as well. The redefined function, (6) is applied in two different ways depending on the parameter $x \in \{0, 1\}$. C_x is a list of values, containing one element for each node covered in an iteration of CEAS (listed in H).

$$C_x[n] = \Big(\sum_{i=0}^{\vartheta_x[n]} \frac{1}{\Theta_x + 1 - i} \Big)^2 \tag{6}$$

For $x = 1$, load samples L are used, accounting for all concurrently executing services on the nodes sampled in L. $x = 0$ in turn represents solely the mappings, M, taking into account the load imposed by the components that are part of a given service. The two usages differ in the upper-bound of the summation and the constant in the denominator, ϑ_x and Θ_x respectively, defined in (7) and (8).

$$\vartheta_x[n] = |m_n| \cdot w + x \cdot L[n] \quad \text{for } x \in \{0, 1\} \tag{7}$$

$$\Theta_x = \sum_{\forall n \in H} \vartheta_x[n] \quad \text{for } x \in \{0, 1\} \tag{8}$$

Θ_x represents the overall execution load of one service ($x = 0$) or all services ($x = 1$). Above, we assume that all replicas have the same weight, denoted w, hence their load can be assessed by multiplying with the number of replicas mapped to a given node, $|m_n|$. Accordingly, it is ϑ_x where parameter M of the cost function is used in this setting. This definition can easily be changed to support individual replica weights. In summary, Θ_0 is the total processing resource demand of one service, whereas Θ_1 accounts for the added load of replicas of *other* concurrent components. In L, we account only for those instances that are mapped to the nodes covered in a given iteration (given in H), and as such have reserved processing power for themselves. This way, the list $C_x[n]$ provides a quadratic approximation of the share of load associated with each node as experienced by CEAS, an approximation only as CEAS does not have an exact global overview over the total offered load. Thus, the overall cost function used for dependability becomes

$$F_3(D, M, H, L) = \frac{1}{|D|} \cdot \sum_{\forall n \in H} C_0[n] \cdot \sum_{\forall n \in H} C_1[n] \tag{9}$$

On the one hand, C_0 applies solely to replicas of one service, this way penalizing the violation of ϕ_2, or in other words, favoring mappings where replicas are not collocated. On the other hand, C_1, is used for general load-balancing and, as such, it takes into account load imposed on nodes by the other services in the network. Using these separate terms we are able to smoothen the output of the cost function used in each iteration, purposefully easing convergence by making

the solution space more fine grained, i.e. simplifying differentiation between very similar deployment mappings with nearly the same cost. Scenarios involving dependability are presented in Sec. 4.3.

2.4 Cluster Costs

In the next scenario, we assume there are different financial costs associated with using different clusters, see for example Sec. 4.4. Hence, we now wish to find deployment configurations that can also minimize the financial cost, while maintaining load balancing and the dependability requirements. To facilitate this, we extend the function in (9) with another term. First, let Λ be the financial cost of using the nodes mapped in M, defined as $\Lambda = \sum_{\forall n_i \in M} |n_i|$, where $|n_i|$ is the financial cost of using node $n_i \in M$. Thus, our new function becomes

$$F_4(D, M, H, L, z) = F_3(D, M, H, L) \cdot (1 + g(z)), \tag{10}$$

where $g(z)$ comes in two variants using a scaling parameter z and Λ

$$g(z) = \begin{cases} z \cdot \Lambda & \text{linear weighting} \\ 1 - e^{-(z \cdot \Lambda)^2} & \text{exponential weighting} \end{cases} \tag{11}$$

Setting $z = 0$ eliminates the financial costs, returning to the original function in (9). The choice of z depends on the cost of using the different clusters. The two alternatives in (11) represent a linear and an exponential increment in financial costs, when $z > 0$. When applying the more fine-grained exponential weighting to the cost function, we expect to observe more balanced mappings, avoiding under-utilization or overloading of clusters.

Next, we present our proposed deployment logic, along with the CEAS optimization method and associated algorithms. The cost functions presented in this section are used in these algorithms to evaluate the deployment configurations.

§3 The Deployment Logic

To solve the deployment mapping problem presented in Sec. 1.1, we use the CEAS method introduced by Helvik and Wittner.[18] CEAS is an agent-based optimization framework, in which the agents' behavior is inspired by the foraging patterns of ants. The key idea in CEAS is to let many agents, denoted *ants*, search iteratively for the solution to a problem taking into account the constraints and a predefined cost function. Every iteration consists of two phases. In the *forward search* phase, ants search for a possible solution, resembling the search for food in real-world ants. The second phase is called *backtracking*, in which ants – after evaluating the solution found during forward search – leave markings, called *pheromones*, that are in proportion to the quality of the solution. Pheromones are then distributed at different locations in the search space and can be used by forward ants in their search for improved solutions. Therefore, the best solution will be approached gradually. To avoid getting stuck in premature and sub-optimal solutions, some of the forward ants explore the

search space, ignoring the pheromones. There is a principal difference, however, between the various existing ant-based systems and the approach taken in CEAS in evaluating the solution and in pheromone updates. CEAS uses the *Cross Entropy (CE) method* for stochastic optimization introduced by Rubinstein.[32] The CE method is applied during the pheromone updating process, gradually changing the probability matrix $\mathbf{p}_r$ according to the cost of the solution found in iteration r. Then, the objective is to minimize the cross entropy between two consecutive probability matrices $\mathbf{p}_r$ and $\mathbf{p}_{r-1}$. For a tutorial on the method,[32] is recommended. Next we present the CEAS method in more detail, followed by a generalized version of our deployment algorithm, which was applied in the evaluations in Sec. 4. Minor algorithmic differences between the different scenarios are also discussed.

3.1 The Cross Entropy Ant System

We apply CEAS to obtain efficient deployment mappings in the form $M : \mathbf{C} \rightarrow \mathbf{N}$ between sets of components, $\mathbf{C}$, and sets of nodes, $\mathbf{N}$. Ants move between nodes across network links in search for nodes with hosting capacities. The cost of mappings is evaluated using the cost functions discussed in Sec. 2, i.e. applying them as $F(M)$ in every iteration of CEAS. The pheromone values, $\tau_{mn,r}$, in CEAS for deployment mapping are assigned to the component set m deployed at node n at iteration r. The *random proportional rule* (**rpr**) in (12) is used to select deployment mappings. That is, during normal forward search, a set of components is selected according to the **rpr** matrix $p_{mn,r}$

$$p_{mn,r} = \frac{\tau_{mn,r}}{\sum_{l \in M_{n,r}} \tau_{ln,r}} \tag{12}$$

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

$$H(F(M_r), \gamma_r) = e^{-F(M_r)/\gamma_r} \tag{13}$$

which is computed for all r iterations such that the expected overall performance satisfies

$$h(p_{mn,r}, \gamma_r) = E_{\mathbf{p}_{r-1}}(H(F(M_r), \gamma_r)) \geq \rho \tag{14}$$

$E_{\mathbf{p}_{r-1}}(X)$ is the expected value of X s.t. the rules in $\mathbf{p}_{r-1}$, and ρ is a *search focus* parameter close to 0 (typically 0.05 or less). 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 $H(F(M_r), \gamma_r)$.

To avoid centralized control and synchronized batch-oriented iterations, the cost value $F(M_r)$ is calculated *immediately* after each sample, i.e. when all components are mapped, and an auto-regressive performance function, $h_r(\gamma_r) = \beta h_{r-1}(\gamma_r) + (1 - \beta)H(F(M_r), \gamma_r)$ is applied. This function is approximated by

$$h_r(\gamma_r) \approx \frac{1 - \beta}{1 - \beta^r} \sum_{i=1}^{r} \beta^{r-i} H(F(M_r), \gamma_r) \tag{15}$$

where $\beta \in \langle 0, 1\rangle$ is a *memory factor*, used for weighting (geometrically) the output of the performance function. The temperature γ_r in turn is determined by minimizing it subject to $h(\gamma) \geq \rho$. The temperature furthermore is equal to

$$\gamma_r = \{\gamma \mid \frac{1-\beta}{1-\beta^r} \sum_{i=1}^{r} \beta^{r-i} H(F(M_i), \gamma) = \rho\} \tag{16}$$

which is a complicated (transcendental) function that is both storage and processing intensive since all observations up to the current sample, i.e. the entire mapping cost history $\{F(M_1), \cdots, F(M_r)\}$ must be stored, and weights for all observations have to be recalculated.[18] This can be a prohibitively large resource demand, especially in online nodes. As a resolution we assume, given a $\beta \approx 1$, that the changes in γ_r are typically small from one iteration to the next, which enables a first order Taylor expansion of (16) as follows

$$\gamma_r = \frac{b_{r-1} + F(M_r) e^{-F(M_r)/\gamma_{r-1}}}{(1 + \frac{F(M_r)}{\gamma_{r-1}}) e^{-F(M_r)/\gamma_{r-1}} + a_{r-1} - \rho \frac{1-\beta r}{1-\beta}} \tag{17}$$

where $a_0 = b_0 = 0$ and $\gamma_0 = -F(M_0)/\ln \rho$. Furthermore,

$$\begin{aligned} a_r &\leftarrow \beta(a_{r-1} + (1 + \frac{F(M_r)}{\gamma_r}) e^{-\frac{F(M_r)}{\gamma_r}}) \\ b_r &\leftarrow \beta(b_{r-1} + F(M_r) e^{-\frac{F(M_r)}{\gamma_r}}) \end{aligned} \tag{18}$$

where the performance function, (13), is adopted. The pheromone values in CEAS are a function of the *entire history* of mapping cost values, hence CEAS has what is denoted a search history dependent quality function.[42] Updates to the pheromone values are made by applying the performance function, (13), combining the last cost value $F(M_r)$ and the temperature γ_r, calculated by (17). Pheromones are updated as follows.

$$\tau_{mn,r} = \sum_{k=1}^{r} I((m,n) \in M_k) \beta^{\sum_{x=k+1}^{r} I((m,\cdot) \in M_x)} H(F(M_k), \gamma_r) \tag{19}$$

The *memory factor*, β, supplies geometrically decreasing weights to the output of the performance function, enabling evaporation of pheromones. The exponent of β is somewhat complex since ants during *backtracking* do not update all nodes in the network, only those nodes that were visited during the preceding forward phase. The exponent in (19) represents the number of ants that have updated node n between time-step r and k when a mapping M_k was found, while $r - k$ is the total number of updates in the system, i.e. total number of ants that returned between time-step r and k. Hence, $r - k \geq \sum_{x=k+1}^{r} I((m,\cdot) \in M_x)$. However, as for (16), excessive processing and storage requirements also apply

for (19). A (second order) Taylor expansion of (19) is appropriate, giving

$$\tau_{mn,r} \approx I((m,n) \in M_r)e^{-\frac{F(M_r)}{\gamma_r}} + A_{mn} + \begin{cases} -\frac{B_{mn}}{\gamma_r} + \frac{C_{mn}}{\gamma_r^2} & \frac{1}{\gamma_r} < \frac{B_{mn}}{2C_{mn}} \\ -\frac{B_{mn}^2}{4C_{mn}} & \text{otherwise} \end{cases} \quad (20)$$

where

$$\begin{aligned} A_{mn} &\leftarrow \beta(A_{mn} + I((m,n) \in M_r)e^{-\frac{F(M_r)}{\gamma_r}}(1 + \frac{F(M_r)}{\gamma_r}(1 + \frac{F(M_r)}{2\gamma_r}))) \\ B_{mn} &\leftarrow \beta(B_{mn} + I((m,n) \in M_r)e^{-\frac{F(M_r)}{\gamma_r}}(F(M_r) + \frac{F(M_r)^2}{\gamma_r})) \\ C_{mn} &\leftarrow \beta(C_{mn} + I((m,n) \in M_r)e^{-\frac{F(M_r)}{\gamma_r}}(\frac{F(M_r)^2}{2})) \end{aligned} \quad (21)$$

The initial values for (21) are $A_{mn} = B_{mn} = C_{mn} = 0$ for all (m,n). For a stepwise explanation of the Taylor expansion and how it is applied, we refer to Appendix A in [37]. Further improvements in the scalability of CEAS are described in [16].

3.2 The Deployment Algorithm

In this section, we present our general deployment algorithm and explain how it is executed. First, the task of obtaining deployment mappings for a given service is assigned a species – a given type – of ants via the service model that can be interpreted by the logic. The service model contains the set of components, $\mathbf{C}$, to be mapped. Ants are emitted (cf. Algorithm 1) continuously and select nodes to visit depending its type. There are two types of ants, explorer and normal ants. *Normal ants* select a subset of $\mathbf{C}$ (can be $\emptyset$) at every node they visit based on the content of the local pheromone table at the node, whereas *explorer ants* select a subset based on a random decision, ignoring the pheromones. The selections made by the ant are stored in M, and carried along with the ant; they represent a deployment mapping made during one iteration of the algorithm.

Initially, only explorer ants are used to explore and cover a significant portion of the mapping problem space by random sampling. The length of initial exploration depends on the problem size, in terms of network size and number of services. After initial exploration, the majority of ants are normal ants, while a smaller fraction are explorers, typically 5-10 percent. This continued exploration is meant to capture fluctuations in the network, e.g. new nodes connecting, and thereby improve responsiveness to dynamism in the environment. The normal ants try to find an optimal mapping.

We also distinguish between two phases in one iteration of the algorithm. In every iteration, the ant starts with *forward search* and completes the iteration with *backtracking*. During *forward search*, the ant obtains a mapping M, i.e. a suggested deployment for the service, which can then be evaluated by $F(M)$. This ends the first phase and *backtracking* can start, which consists of revisiting the nodes visited in the first phase – according to the hop-list H – and updating the pheromone databases in those nodes. This ends an iteration and a new ant

can be emitted, starting a new iteration of the algorithm, unless a stopping criteria is met. Usually however, the algorithm will continue to explore the network for improved mappings; enabling adaptation to changes (reconfiguration) in the execution context.

Generally, we have a trade-off between convergence speed and solution quality. Nevertheless, while deploying services in a dynamic environment, a premature solution that satisfies both functional and non-functional requirements often suffices. ACO systems have been proven to find the optimum at least once with a probability close to one, and after that convergence to the optimum is secured in a finite number of iterations.[34] Since CEAS can be considered as a subclass of ACO, the optimal deployment mapping will eventually emerge.

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

```
1: Initialization:
2:    r ← 0                                              {Number of iterations}
3:    γ_r ← 0                                            {Temperature}
4: while ∞                                               {Stopping criteria can be applied here}
5:    M ← antAlgo(l, k)                                  {Emit new ant for service l from Nest k, obtain M}
6:    update(availableClusters)                          {Check the number of available clusters}
7:    if splitDetected() ∨ mergeDetected()
8:       release(S_l)                                    {Delete existing bindings for all instances c_i ∈ C_l}
9:    if Φ(M, availableClusters)
10:      bind1(M)                                        {Bind one of the still unbound instances in C_l}
11:   r ← r + 1                                          {Increment iteration counter}
```

Improved dependability of the approach can be obtained by means of replicated ant *nests*. The same ant species may be emitted from multiple nests, providing resilience to node failures affecting the node hosting a nest. Note that, this nest replication does not lead to flooding of the network with ants, as the rate of ant emission in a stable network can be divided equally among nests. Moreover, ants emitted from different nests, but associated with the *same service*, will operate on the *same pheromone table entries* in the nodes they visit. During execution of CEAS, synchronization between nests is not necessary. However, a primary nest must make the final deployment decision, triggering physical placement of the components.

In contrast to CEAS used for routing, where the temperature is stored in the destination node, our CEAS implementation has no notion of destination for the deployment mapping. Instead a mapping, M, is distributed over a set of nodes. Yet, ants are able to find the same mapping M, while visiting the same set of nodes, possibly in a different order, and making the same mapping decision. To provision for this capability, ants returning to their nest at the end of the *backtracking* phase, will pass on the temperature parameter to their immediate successor ants.

We clearly distinguish between the notions of component *mapping*, *binding* and *deployment*. The *mapping* M is a variable list constantly optimized, iteration by iteration by the logic only visible internally to the algorithm. When an instance is *bound* to a node that means that the particular mapping for that

Algorithm 2 Ant code for mapping service S_l

```
1: Initialization:
2:    H_r ← ∅                                   {Hop-list; insertion-ordered set}
3:    M_r ← ∅                                   {Deployment mapping set}
4:    D_r ← ∅                                   {Set of utilized clusters}
5:    L_r ← ∅                                   {Set of load samples}
6: function antAlgo(r, k)
7:    γ_r ← Nest_k.getTemperature()             {Read the current temperature}
8:    foreach c_i ∈ C                           {Maintain bound mappings}
9:       if c_i.bound()
10:         n ← c_i.boundTo()                  {Jump to the node where this comp. is bound}
11:         n.reallocProcLoad(S_l, e_i)        {Allocate processing power needed by comp.}
12:         l_{n,r} ← n.getEstProcLoad()       {Get the estimated processing load at node n}
13:         L_r ← L_r ∪ {l_{n,r}}              {Add to the list of samples}
14:   while C ≠ ∅                               {More instances to map}
15:      n ← selectNextNode()                   {Select next node to visit}
16:      if explorerAnt
17:         m_{n,r} ← random(⊆ C)               {Explorer ant; randomly select a set of comps.}
18:      else
19:         m_{n,r} ← rndProp(⊆ C)              {Normal ant; select comps. according to (12)}
20:      if {m_{n,r}} ≠ ∅, n ∈ d_k              {At least one comp. mapped to this cluster}
21:         D_r ← D_r ∪ d_k                    {Update the set of clusters utilized}
22:      M_r ← M_r ∪ {m_{n,r}}                  {Update the ant's deployment mapping set}
23:      C ← C − {m_{n,r}}                      {Update the set of replicas to be deployed}
24:      l_{n,r} ← n.getEstProcLoad()           {Get the estimated processing load at node n}
25:      L_r ← L_r ∪ {l_{n,r}}                  {Add to the list of samples}
26:   cost ← F(D_r, M_r, H_r, L_r)              {Calculate the cost of this given mapping}
27:   γ_r ← updateTemp(cost)                    {Given cost, recalculate temperature according to (16)}
28:   foreach n ∈ H_r.reverse()                 {Backtrack along the hop-list}
29:      n.updatePheromone(m_{n,r}, γ_r)        {Update pheromone table in n}
30:   Nest_k.setTemperature(γ_r)                {Update the temperature at Nest_k}
```

instance is not changed anymore by the ants until that binding is erased again. Lastly, by *deployment*, we refer to the physical placement and instantiation of an instance on a node, which is triggered after the mapping M for the given service has converged to a satisfactory solution. The latter property ensures that there is no undesirable fluctuation in the migration of replicas using our method.

Improving convergence is the concept of *binding* of components, which allows nests to *fix* one instance in the latest mapping M obtained by the ants, if some condition applies. For example, we have a condition that checks if the mapping M satisfies Φ as condition of a *bind* event. After a *bind*, ants for the same service do not change the fixed mapping in subsequent iterations and new searches will be conducted for the remaining instances only. Importantly however, bound instances are also taken into account when the cost of the total mapping is evaluated. Should a split or a merge event occur in the network, these bindings are erased in the ant nest and the total amount of instances will be taken into consideration in the following searches. In Algorithm 1 and Algorithm 2, we present the version of the deployment algorithm that uses binding as well as guided random hopping (Algorithm 3), which we discuss next.

First, an ant visits the nodes, if any, that already have a bound instance

mapped to, in order to maintain these mappings. These will also be taken into account when the cost of the total mapping is evaluated. The pheromones corresponding to bound mappings will also be updated during *backtracking*. Ants allocate processing power corresponding to the execution costs of the bound replicas, derived from the service specification. This first phase of an ant's tour is denoted *maintenance*. After this phase, ants turn to guided random hop-selection.

The 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. Pheromone tables are used only for selecting components or replicas to map to nodes. The advantage of using the guided selection shown in Algorithm 3 as opposed to a pure random walk lies in that with the proper guidance, the frequency of finding an efficient mapping is higher. In case of replica management, for example, idea is that at first the next node is selected from a cluster that has not yet been utilized until all visible clusters are covered, leading to better and faster satisfaction of ϕ_1 (see Sec. 2). Then, next hop selection continues with drawing destinations from the set of nodes not yet used in the mapping. This can be done by checking with the variable M, before reverting to totally random drawing.

Algorithm 3 Next-hop selection procedure for an ant

```
1: function selectNextNode()                          {Guided random jump}
2:   if H = N                                         {All nodes visited}
3:     n ← random(N)                                  {Select candidate node at random}
4:   else
5:     if D = D                                       {All available clusters utilized}
6:       n ← random(N \ M)                            {Select a node that has not been used yet}
7:     else
8:       d_i ← random(D \ D)                          {Select a cluster not yet used}
9:       n ← random(d_i)                              {Select a node within this cluster}
10:  H ← H ∪ {n}                                      {Add node to the hop-list}
11:  return n
```

The guided hop-selection algorithm can be used to different extents, i.e. in case of simple settings without replication management where the network is not partitioned into clusters this function simply can be reverted to random drawing, e.g. the scenarios presented in Sec. 4.1 and 4.2. Whereas when replication is present and dependability rules, Φ, have to be satisfied, the guidance and taboo-listing can be turned on. Next, we discuss how the data in pheromone tables can be represented efficiently.

3.3 Encoding Pheromone Values

Optimization governed by the cost function starts with aligning pheromone values with the sets of deployed components and defining the structure of the pheromone database for the ants. With the underlying set of nodes, each ant will form $|\mathbf{N}|$ discrete sets from the set of available components ($\mathbf{C}$) that need to be deployed and will evaluate the outcome of that deployment mapping at the

end of each iteration. In the simplest encoding, we assign a flag to each of the component instances and build a bitstring for a service of size $2^{|\mathbf{C}|}$. This way a single pheromone database instance located at a node becomes equal in size to the number of possible combinations for forming a subset of $\mathbf{C}$. After normalizing the pheromones in a node we can observe the probability distribution of component sets mapped to that particular node by the ant system. Eventually, after convergence the suggested solution emerges in the distributed pheromone database with probability near to one.

In case of a *normal* ant, the selection process for selecting a set of instances to map to a node depends on the form of the pheromone tables, in particular on how the pheromone values are encoded. Appropriate solutions can be found using different encodings, however, there are differences in terms of convergence times and solution quality. Efficient encodings are required for scalability of the logic as well. In [8], we proposed and evaluated three different pheromone encodings. Generally, pheromone tables can be viewed as a distributed database with elements located in each node available in the network considered for deployment. Entries in the database have to be able, on one hand to describe arbitrary combinations of components or replicas. On the other hand, as the distributed database consumes some memory in every node – and the required memory grows both with the amount of services deployed ($|\mathbf{S}|$) and with the sizes of the services ($\mathbf{C}$) –, the size of this database is crucial for scalability. We wish to accommodate as many services as possible, thus we have to efficiently manage the memory need, which we can directly influence by choosing an appropriate pheromone encoding. Beside the storage needs, an individual ant agent has to browse through the pheromone entries during its visit to a node. Clearly, a more compact pheromone database helps speeding up execution of the tasks it has to perform. The different encodings we proposed and their corresponding sizes are shown in Table 2.

Table 2 Three Pheromone Encodings for a Service with $|C|$ Instances

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

The first encoding, called *bitstring*, is the largest as it holds a single value for all possible combinations of replica mappings in every node, which can result in prohibitively large memory need. For example, in case of 20 components 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 reduce the table size, we can apply simpler bookkeeping taking into account solely the number of replicas mapped to a given node, shown in *# replicas*. This is the most compact pheromone entry encoding, however, the tradeoff is that it cannot distinguish between replicas in a service specification, thus it can only be applied if there is no need to distinguish between component replicas, e.g. due to equally sized replicas. As a good compromise in between, we

have developed the *per comp* encoding that results in no information loss, while still being linear in size. The *per comp* encoding uses one distinct pheromone entry for every instance indicating whether or not to deploy them at a given node. The slight disadvantage is that ants arriving at a node have to decide on the deployment mapping of each replica, one-by-one reading the multiple pheromone entries corresponding to the elements of the service (one separate table element for each). Nevertheless, this encoding provides the necessary reduction in database structure size for allowing scaling up to larger amounts of services and larger service sizes.

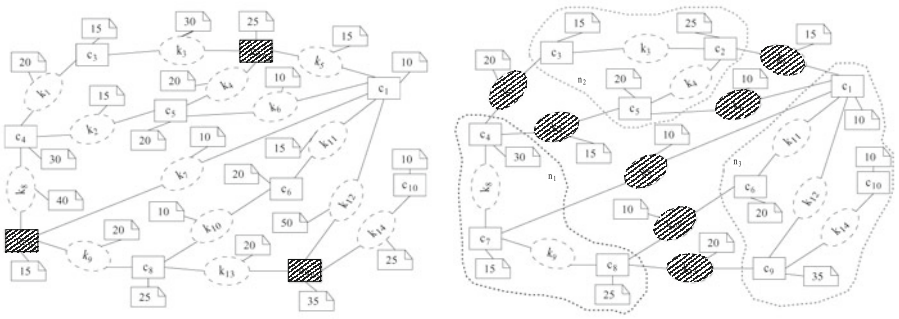
In the following, examples are presented where CEAS and the deployment logic is applied in different scenarios.

§4 Example Scenarios

To demonstrate the deployment mappings that can be obtained using our deployment logic, 4 different scenarios are presented in the subsequent subsections, followed by a subsection on cross-validation of some of the results by application of a centralized method.

4.1 Deployment of Collaborating Software Components

As a first example, we consider a scenario that has been introduced by Efe, originally modeling clustering of modules and cluster assignment to nodes.[12] The same scenario has been investigated by several authors, including Widell et al. who compared the related results in [36]. This artificial clustering problem is modeled as a collaboration of components and is used to test the deployment logic. We define Efe's example as a collaboration of $|\mathbf{C}| = 10$ components (labelled $c_1 \dots c_{10}$) to be deployed and $|\mathbf{K}| = 14$ collaborations between them $(k_1, \dots k_{14})$, as depicted in Fig. 4(a). Besides, the execution and communication costs, we have a restriction on components c_2, c_7, c_9, regarding their location. They must be placed in nodes n_2, n_1, n_3, respectively.



(a) Collaborations and components in the example

(b) Optimum with 3 nodes

Fig. 4 Simple Example Service

In this example, the target environment consists of $|\mathbf{N}| = 3$ identical, interconnected nodes. To gather information continuously, the ants employed by the logic sample the CPU load levels, $\hat{l}_n$. We target minimum remote communication, i.e. we take into account communication costs for collaborations between two components if they are not collocated in the same node. At the same time, we look for a globally balanced CPU load over all the nodes available. Furthermore, for this first example, we have $T \cong 68$ (cf. (1)) as average target load in the 3 nodes.

The optimum solution of the example is depicted in Fig. 4(b), with the lowest possible cost value of $17 + 100 = 117$ (cf. (3)). Finding an efficient deployment with the lowest cost is illustrated in Fig. 5, with absolute values on the Y-axis. After an initial 2000 *explorer* ants, optimization starts and the overall cost is converging to the optimum value of 117. The size of the dynamically allocated pheromone database – in which a threshold can be applied to regulate the amount of significant entries – can also be observed in the bottom half of the figure. The database size tops at 2^7 as the number of maximum available components is 7 out of the total of 10, with 3 bound components. In case of convergence to a single solution, like in the example at hand, solutions other than the optimum can be evaporated from the database, thus reducing its size if needed, as shown.

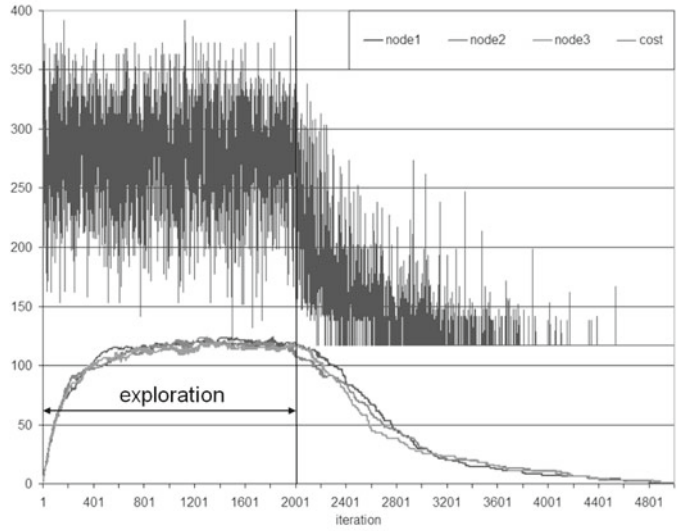


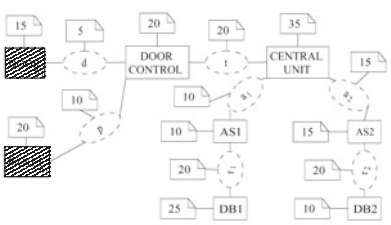
Fig. 5 Observed cost and pheromone database sizes

For more details and comparison of the results obtained with the CEAS and other methods using this example, we refer the reader to [6]. Accordingly, we find that our distributed approach is capable of obtaining efficient mappings in NP-hard deployment scenarios. Next, we continue with increasing complexity by considering more services simultaneously in another example setting.

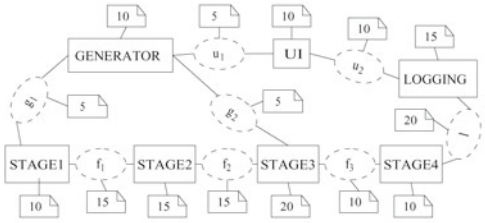
4.2 Multiple Species

In [7], we introduced three service models for experimental use. The first example has been introduced originally in [25]. S_1 operates a security door and a card reader with a keycode entry panel using authentication and authorization servers and databases. The second example models a video surveillance system. A central control with a recording unit manages the system and uses a main and a backup storage device for storing surveillance information. S_3 is a model of a process controller that consists of 4 main stages of processing, logging, a user interface, and a generator component. The service models are presented in Fig. 6.

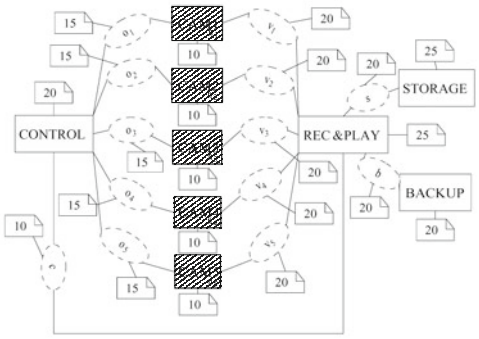
An ant species is assigned to each of the services and deployment mappings are obtained using a network of 5 nodes. This example has multiple optimal and near-optimal solutions with different sets of components deployed on various nodes. In [7], we tested the effectiveness (e.g. number of iterations required for convergence, average cost of mappings found) of the approach considering that in the multiplicative cost function, (4), we omitted any global knowledge, i.e. we do not use the global shared knowledge T, (1), anymore. Instead, the parallel species are aware of each other' processing power demand through sampling, i.e.



(a) S_1



(b) S_3



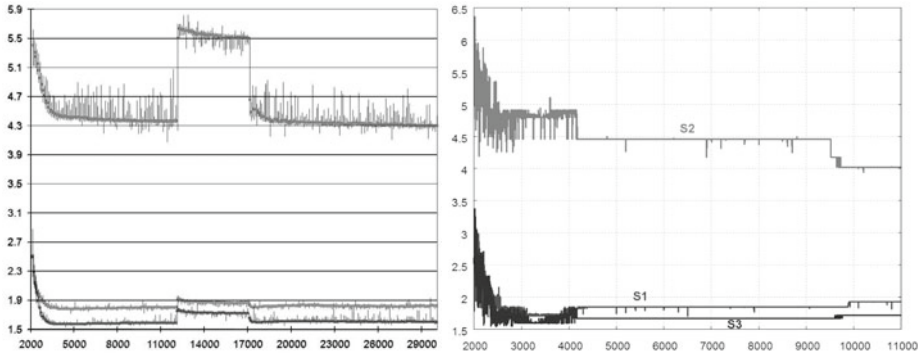
(c) S_2

Fig. 6 3 Service Examples

L. We have found that the multiplicative function (4) allows the logic to deploy multiple parallel species equally effectively. Synchronization of the separate ant nests is not necessary, however, only one designated nest shall be allowed to trigger placement.

To evaluate adaptation capabilities of the logic, we investigated two simple scenarios, a single node failure and reparation, and the appearance of a new node in the network. The new node is added after all the species have converged to a solution with 5 nodes. The evolution of costs (Y-axis) – obtained by (4) – for the 3 services in the example is shown in Fig. 7 as a function of the number of iterations (X-axis). A node error is injected after iteration 12000 followed by a repair approximately 4000 iterations later (Fig. 7(a) shows average and min-max values). The first 2000 iterations represent the initial *exploration* phase. Due to the abrupt change in the context – a previously used node disappears – costs increase quickly after the failure. Service S_2 suffers most, as indicated by the highly increased costs, mostly due the large communication demand of that service. Deployment costs return to normal again somewhat slower after the node has been repaired and explorer ants discover the new, more useful configuration.

In the second test, a new (6th) node is added to the network of 5 nodes after around 5000 iterations, at which point the 3 species are already converged to a stable solution. At this point, 5% of the emitted ants are *explorers*, which eventually (approx. after 9500 iterations) discover the new node and a new, more efficient deployment (Fig. 7(b)). It is also to be noted that the species agree on a new configuration where services S_1 and S_3 suffer some degradation, i.e. in terms of higher costs, but S_2 has a high gain, thus the overall utility of the new deployment is better. Regarding the number of iterations, we note that using conventional exhaustive search methods, we would have needed to explore $|\mathbf{N}|^{|\mathbf{C_{S_k}}|}$ possible configurations for a service S_k and we have to add up the numbers for $\forall S_k \in \mathbf{S}$ to represent the problem size. So, first we gain on



(a) Node error/repair (b) New node appearing

Fig. 7 Costs for the 3 Services in Two Test Scenarios

splitting the problem of deploying multiple services using one type of species for each service simultaneously. Second, we increase computational effectiveness by not having to explore the search space – e.g. of size 5^{24} possible configurations in this example with 3 services – exhaustively.

4.3 Deployment of Component Replicas

We focus next on dependability achieved by efficient replication management. Fig. 8 depicts a simple example service model. A component is actively replicated in 4 replicas, R_1 to R_4, with the continuous updating mechanism modeled as collaborations between the replicas. Each replica in the service performs according to the client requests, thus, replicas have the same execution cost.

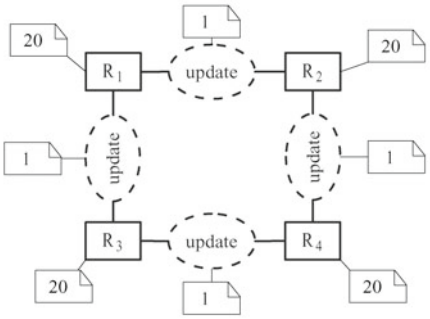
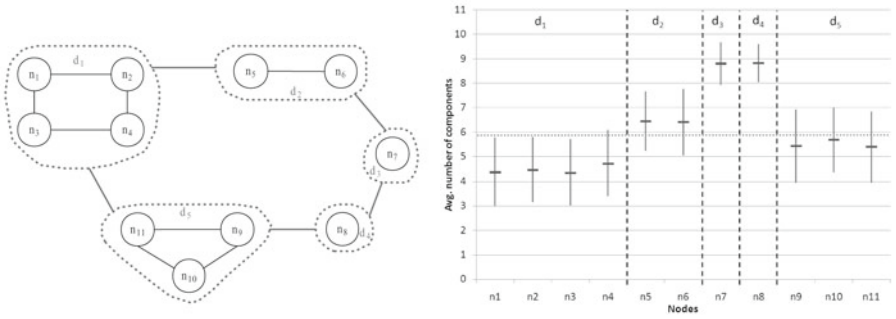


Fig. 8 Replicated Components in Example Service S_1

The example, furthermore, consists of 10 services ($S_1 \ldots S_{10}$) that are being deployed, using 10 independent species of ants. Also, we use 20 ant nests to be able to look at a simple cluster splitting and merging scenario, with one nest remaining for each of the species in each of the regions formed after the split. Each service S_i, $i = 1 \ldots 10$ has a redundancy level (amount of replicas) of $i + 1$. The network of nodes for the experiment consists of 11 fully interconnected nodes, partitioned into 5 clusters (Fig. 9(a)). Simulations of the scenario were conducted in a discrete event simulator custom built and programmed in Simula/DEMOS language.

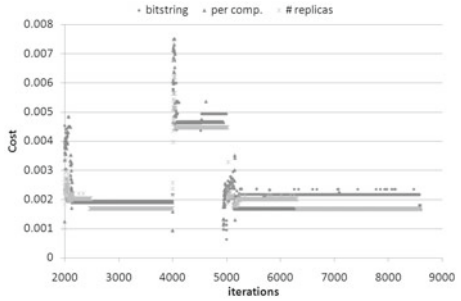
First, we look at the objective of obtaining a balanced deployment mapping with respect to execution load. This objective has to be followed while maintaining the dependability rules of Φ. In Fig. 9(b), we look at the average number of replicas mapped to the 11 nodes in the test network. A total amount of 65 identically sized component replicas are mapped, which – in a homogenous, non-clustered network – would give an average of 5.91 replicas per node (shown as a dotted horizontal line). As an effect of cluster disjointness (ϕ_1), smaller clusters, such as d_3, d_4, suffer from overloading, but generally replicas are placed quite evenly across the available nodes, showing that cooperation between the species works. Furthermore, for a larger experiment with 50 nodes and 275 replica instances we refer to [8].

In the same example setting, we conducted simulations to test adaptation



(a) Test network of nodes clustered into 5 clusters

(b) Load-balancing over 11 nodes



(c) Splitting/merging of cluster d_5

Fig. 9 Example Scenario with 10 Services

capabilities of the logic. Three different pheromone encodings are tested in this example, more about the various encodings and their effects in Sec. 3.3. To test capabilities to remedy cluster splitting and merging, cluster d_1 containing 4 nodes is split from the rest of the network at the cluster boundary and some time later, it merges back, thus restoring the original network scenario. For example, the cost output in case of service S_{10} is displayed in Fig. 9(c). Cluster d_1 splits from the rest after iteration 4000 and we can observe how the swarm adapts and obtains new mappings for a more expensive configuration (increased cost) due to the reduced network. Similarly, after the merge of the two regions, around approximately iteration 5000, mappings are adapted utilizing cluster d_1 again, thus resulting in a lower cost configuration again.

Regarding the number of iterations required for obtaining reasonable and stable mappings, e.g. shown on the X-axis in Fig. 9(c), we conclude that they are reasonably low and do not increase the overhead significantly, especially compared to exhaustive search, which would require evaluation of 11^{65} possible configurations. Mapping of replicas considering the dependability rules becomes

harder when the number of replicas in a service is close to the amount of available nodes. Instead of having hard-constraints that strictly cannot be violated, like e.g. in traditional integer programming, we utilize soft-constraints incorporated into the cost functions we use. The tradeoff might be that sometimes the algorithm prefers a globally lower cost mapping with better overall load-balancing that, however, violates some of the soft-constraints for one service, e.g. for a large service that has as many replicas as many nodes exist, there might be a single collocation (violation of ϕ_2) due to the better utilization of an otherwise under-utilized node.

Besides looking at adaptation, in Fig. 9(c), we also present how the costs evolve using the three different encodings introduced in Sec. 3.3. After a split and merge event the *bitstring* encoding converges to a solution with slightly higher overall cost than before, whereas the lowest cost is obtained first by *per comp.* and somewhat later by *# replicas*. The first 2000 iterations are not shown as, initially, a random cost figure appears corresponding to exploration that is omitted here. Every simulation starts with 2000 explorer iterations for the sake of comparability, even though the amount of initial exploration was constrained by the *bitstring* encoding. The more compact encodings require significantly less iterations, e.g. one tenth of the amount used. 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.

Table 3 Success Rates of the Three Encodings in the Example Setting

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

Considering the dependability rules $\phi_1 \wedge \phi_2$, Table 3 shows the three different pheromone encodings and the percentage of test cases, which succeeded in satisfying the two rules. The results were obtained by executing the algorithm 100 times for each encoding, with different input seeds. The results indicate that choosing *per comp.* not only provides the best compromise between scaling and descriptiveness but gives more efficient results too.

4.4 VM Instance Placement in Private and Public Clusters

In this section, we look at how self-organizing techniques applied for system (re)configuration can be used to improve scalability and dependability of virtualized service systems. Specifically, we discuss deployment of virtual machine (VM) images to physical machines in a large scale network. Simulation results with the decentralized deployment logic are presented for an example cloud computing scenario. For additional details about the deployment scenario, see [9].

To demonstrate the behavior of the logic, consider the scenario depicted in Fig. 10. The network consists of 5 private clouds (Cloud $C, \ldots, G$) connected

to public cloud providers (Cloud A, B) via the Internet. The publicly available capacities can be utilized on demand, but are subject to financial costs. Conversely usage of private clusters is free for a service with a home location in that private cloud. Thus, deploying and hosting a VM instance on a node within one of the clouds implies additional costs; namely, a cost of 10 for a node in *Cloud A*, 1 for *Cloud B* and 0 cost for the private clouds $C \ldots G$. The scaling parameter, cf. (11), we applied in the example was $z = 0.1$.

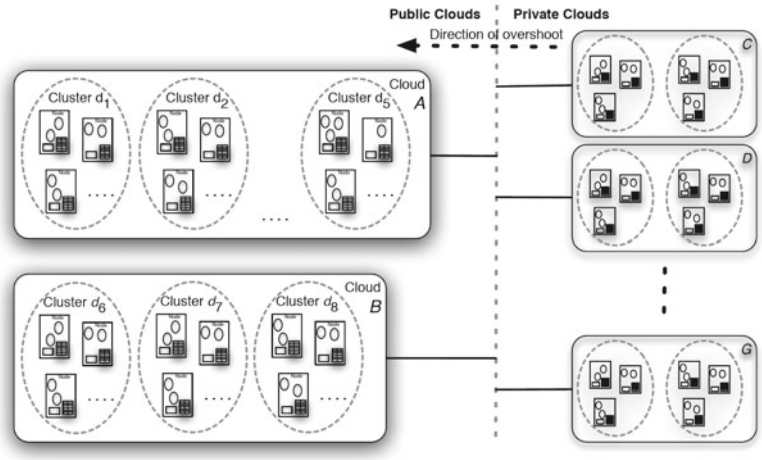


Fig. 10 Cloud Computing Example Scenario

Intuitively, this partitioning and cost assignment models a scenario where organizations naturally execute VM instances within their privately owned cluster as long as the requirements allow, i.e. satisfactory replication levels can be achieved with the available resources in the private clusters. Accordingly, hosting VMs in private clusters is considered free as opposed to hosting in public clouds. Moreover, in the example, a trade-off exists 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 less resources.

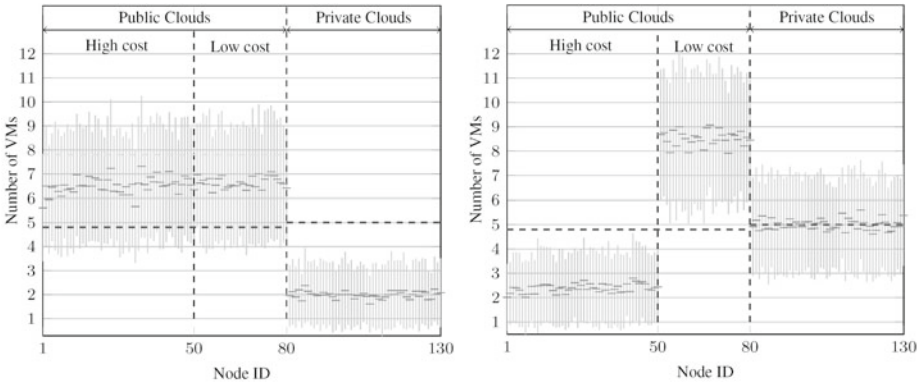
Simulations execute the deployment task of mapping 125 services, i.e. 25 in each private cloud, using public clouds for deployment on demand, without allowing usage of resources in private clusters other than the originating ones. This is practically achieved by assigning ∞ cost for neighboring private clouds. Each of the 125 services consist of 5 VM instances – among others for dependability reasons – that have to be deployed, thus resulting in the task of deploying a total number of 625 VMs.

The target network of the private and public clouds offers 130 nodes in different clusters. 5 and 10 nodes are available in each private and public cluster respectively. Any single authority owning a private cloud administers two clusters, which together with the 5 (cloud A) plus 3 (cloud B) clusters result in a total of 18 clusters. Regarding the services one species is used for each, giving

25 nests in each private cloud emitting ants. We investigate 2 combinations of (10) and (11) with the above example setting:

1. no cloud costs, $z = 0$;
2. exponential weighting for cloud costs, $z > 0$.

We conducted simulations with the above variants of cost evaluation and checked the resulting deployment mappings.



(a) VMs per node, $z = 0$, no weighting

(b) VMs per node, $z > 0$, exponential weighting

Fig. 11 Example Scenario with 10 Services

In Fig. 11(a), mapping of VM instances – after convergence – is shown when $z = 0$, i.e. every node n_i has zero financial cost for hosting a VM, error bars show the deviation of the results. In case there are no financial costs of using a node, we can observe that VMs are mapped evenly – while maintaining the dependability requirements – over all the nodes resulting in an average of 2 VMs per node in the private clouds, which leaves $625-(5\cdot20) = 525$ VMs for the public clouds A and B. Further, the VMs in the public clouds are mapped quite evenly too over the available 80 nodes around the average of $525/80 = 6.6$. Moreover, the logic does not distinguish between the two different public cloud offerings in this case. The two extremes of mapping 3 VMs in public ($(3 \cdot 125)/80 = 4.7$) and 2 in private clusters; and mapping all 5 VMs in the services in the public clusters ($(5 \cdot 125)/80 = 7.8$) are shown by the dashed lines.

When $z > 0$, we present the average amount of VM instances per node in case of the exponential cost function in Fig. 11(b). In this case, the deployment logic finds mappings that successfully take into account the financial costs of hosting in public clouds. The public cloud with plenty of resources but, thus, higher costs (Cloud A, nodes $n_{1\ldots50}$) receives a low amount of instances, whereas the lower cost public offering (Cloud B) is heavily loaded with VMs. Note that, the requirements of node and cluster-disjointness are still satisfied. In the private clusters, 5 VMs are mapped to each node on average, i.e. each one of the 25

services places 1 VM in each of the two locally available clusters, which incur 0 additional cost. However, due to the cluster-disjointness criteria the 3rd, 4th and 5th VM instance has to be placed to a public cloud. To handle this overshoot, the algorithm looks for the lowest possible increment in cost that gives the resulting deployment. Differences in using the two variants of (11) lay in that with a more complex cost evaluation (exponential instead of a simple linear) more balanced deployment mappings are obtained, i.e. given the cost values assigned to cloud A and B, 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.[9)]

The number of iterations the algorithm requires to produce the results discussed above depends on the problem size. We used 2000 *explorer* ants followed by an additional 3000 (10% *explorer* and 90% *normal*) ants for each species. An increased amount of nodes in itself would not make the deployment problem more difficult to solve. In fact, an increased network size actually allows the algorithm to find better mappings, with lower costs, easier due to the larger amount of available resources. Scalability is impacted to a larger extent by the amount of services and the amount of VMs within the services executed simultaneously, as the number of species executed in parallel is proportional to the number of services and the complexity of the tasks an ant has to perform increases as the number of instances grows.[8)]

With the example presented in this section, we have shown that the deployment logic can be applied in a cloud computing scenario by carefully looking at the cost functions driving the optimization and adjusting them to the perceived utility of the various configurations. In this way, concepts of financial costs in connection with resource usage can also be used in the evaluation of deployment mappings, in addition to the traditional non-functional requirements of performance and dependability.

4.5 Cross-validation of Deployment Mappings with ILP

Distributed execution of our deployment mapping algorithm has been an important design criteria to avoid the deficiencies of existing centralized algorithms, e.g. performance bottlenecks and single points of failure. In addition, we intend to conserve resources by eliminating the need for centralized decision-making and the required updates and synchronization mechanisms. In Sec. 4.1, we presented an example – well-known in task assignment problems – converted to our context of collaborating components (see Fig. 4 for the service and the optimum mapping). In this section, we extend on this initial example with two additional service models, present an Integer Linear Program (ILP) able to solve component deployment problems with load-balancing and remote communication minimization criteria. We then compare simulation results obtained by executing the deployment algorithm on the example models in this section with the optimum cost solutions given by the ILP. Complexity of the deployment examples remains NP-hard, even if we only deal with a single service at a time.

Beside the first model, the second example has an extended solution space,

obtained by extending the first example into a larger service model (15 components, 5 bound, additional collaborations) and increasing the number of nodes (4 nodes) available for deployment mapping. The third example leaves the cardinality of **C** unchanged, while changing the model – the configuration of components – and increasing the amount of collaborations, and the number of available nodes is increased to 6 as well. The concrete amount of components, $|\mathbf{C}|$, and collaborations, $|\mathbf{K}|$, in the service models and the amount of nodes, $|\mathbf{N}|$, are shown in Fig. 12. The ILP we have developed to validate our simulation results will be presented next. We take into account the two counteracting objectives of load-balancing and remote communication minimization and solve the ILP using regular solver software, for further details we refer to [5].

We start with defining the solution variable $m_{i,j}$ representing the set of mappings M.

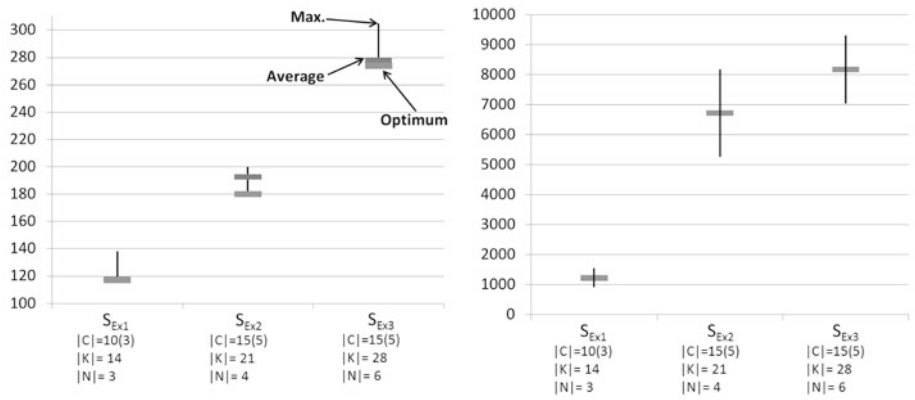
$$m_{i,j} = \begin{cases} 1, & \text{if component } c_i \text{ is mapped to node } n_j, \\ 0, & \text{otherwise.} \end{cases} \tag{22}$$

that will indicate and efficient mapping of components to nodes; and we continue with two parameters. First, $b_{i,j}$

$$b_{i,j} = \begin{cases} 1, & \text{if component } c_i \text{ is bound to node } n_j, \\ 0, & \text{otherwise.} \end{cases} \tag{23}$$

which enables the model to fix some of the mappings, if any components are bound in the model. Second, T, defined in (1) to approximate the ideal load-balance among the available nodes in the network.

Beside the binary $m_{i,j}$, we utilize two additional variables. The first for



(a) Mapping costs (b) Number of iterations

Fig. 12 Cross-validation of the Simulation Results

checking collocations, in col_i.

$$col_i = \begin{cases} 0, & \text{if } c_l, c_k \in k_i \text{ and } c_l \text{ is collocated with } c_k, \\ 1, & \text{otherwise.} \end{cases} \tag{24}$$

Another variable, Δ_j, to indirectly calculate the deviation from the ideal load-balance among the nodes hosting the components.

$$\Delta_j \geq 0, \forall n_j \in \mathbf{N} \tag{25}$$

The objective function used in the ILP is naturally very similar to the linear cost function (3).

$$min \sum_{j=1}^{|\mathbf{N}|} \Delta_j + \sum_{i=1}^{|\mathbf{K}|} f_i \cdot col_i \tag{26}$$

After having established the objective function, the constraints the solutions are subjected to have to be defined. First, we stipulate that there has to be one and only one mapping for all of the components.

$$\sum_{j=1}^{|\mathbf{N}|} m_{i,j} = 1, \forall c_i \in \mathbf{C} \tag{27}$$

The ILP has to take into account that some component mappings might be restricted (*binding*) to particular nodes. Thus, we restrict the variable $m_{i,j}$ using $b_{i,j}$.

$$m_{i,j} \geq b_{i,j}, \forall c_i \in \mathbf{C}, \forall n_j \in \mathbf{N} \tag{28}$$

We introduce two additional constraints to implicitly define the values of the variable Δ_j that we apply in the objective function. We use two constraints instead of a single one to avoid having to use absolute vales (i.e. the *abs*() function) and thus, we avoid non-linear constraints.

$$\sum_{i=1}^{|\mathbf{C}|} e_i \cdot m_{i,j} - T \leq \Delta_j, \forall n_j \in \mathbf{N} \tag{29}$$

$$T - \sum_{i=1}^{|\mathbf{C}|} e_i \cdot m_{i,j} \leq \Delta_j, \forall n_j \in \mathbf{N} \tag{30}$$

Similarly, we define two additional constraints for implicitly building the binary variable, col_i, indicating collocation of components.

$$m_{i,j} + m_{k,j} \leq (2 - col_l), k_l = (c_i, c_k) \in \mathbf{K}, \forall c_i, c_k \in \mathbf{C}, \forall n_j \in \mathbf{N} \tag{31}$$

$$m_{i,j1} + m_{k,j2} \leq 1 + col_l, k_l = (c_i, c_k) \in \mathbf{K}, \forall c_i, c_k \in \mathbf{C}, \forall n_{j1}, n_{j2} \in \mathbf{N} \tag{32}$$

Using the above definitions, the ILP can be executed using an appropriate solver. By submitting the appropriate data, defining the example services introduced

above, we obtain the optimum mappings and their corresponding costs according to the objective/cost function (26), subject to the constraints in (27) – (32).

As a result of the cross-validation, we obtain the absolute minimum cost values (*Optimum*) in the three example settings $S_{Ex1...3}$. Simulation results are generated by executing the algorithm 100 times for each example model. Average costs (*Average*) and the maximum deviation ($Max.$) are shown in Fig. 12(a) as well as the optimum obtained by ILP. Results show that our algorithm finds the optimum 99% of the time in case of the first example. In the somewhat larger scenario, S_{Ex2}, it is more difficult to find the absolute lowest cost mapping, thus we observe larger deviations in mapping costs. However, it is to be noted that by changing the mapping of a single component from the optimum configuration to a near-optimal one increases the costs by values larger than 1, which is also the reason for increased deviations in this case. In fact, the two most frequent sub-optimal solutions, beside the optimum cost of 180, were configurations with a cost of 195 and 200, giving an average of 193 in the end, shown in Fig. 12(a). In case of the third example, S_{Ex3}, the algorithm managed to obtain solutions with costs closer to the absolute optimum – obtained by the ILP – with less deviation at the same time. The main reason for this is that communication costs in S_{Ex3} are relatively more fine grained, which resulted in finding near-optimum solutions with slightly higher costs only. The average number (and deviation) of *normal ant* iterations required by the algorithm to obtain the solutions are shown in Fig. 12(b).

It is to be noted that the difference in complexity is significant between the original example from Sec. 4.1 and the two extended models. Using the simplest binary pheromone encoding (cf. Sec. 3.3), the first example requires a pheromone database of size $3 \cdot 2^7$ in the network of 3 nodes, as the number of unbound components is 7. In the larger examples, the pheromone database size increases to $4 \cdot 2^{10}$ and $6 \cdot 2^{10}$ for CEAS. It is difficult to precisely compare the computational effort required by the ILP and CEAS for solving the same problems. One iteration of a centralized logic with global knowledge, e.g. an ILP, can not really be compared to one iteration in the distributed CEAS, which is a tour made by the ant.

The solver software for the ILP provides some information regarding the iterations and cuts that were required during execution, i.e. 86, 495 and 1075 *simplex iterations*; and 0, 5 and 33 *branch and cut nodes* were reportedly required for the examples S_{Ex1}, S_{Ex2} and S_{Ex3} respectively. The number of required (*explorer* and *normal*) iterations in CEAS is naturally higher than what is required for the ILP with a global overview. However, we advocate that we gain more by the possibility of a completely distributed execution of our algorithm and also because of the capability of adaptation to changes in the context, once the pheromone database is built up after the initial phase.

§5 Related Work

Influencing the software architecture by changing the deployment configuration has been found to be an efficient way to improve utility of services.

Deployment decision making requires an optimization method to function properly and autonomicity has to be built in as a basic functionality. An algorithm has been devised in [26] that is based on calculating the usefulness of alternative configurations as weighted sums. Nevertheless, the resulting approach is not computationally effective and serves as a trial to show that deployment decision making is important and necessary to apply. Various other approaches have been followed to tackle the problem of efficiently mapping instances to resources, or hosts for adequate execution. Many authors start with centralized observation and control, utilizing for example binary integer programming,[3] graph cutting,[20] or some hybrid approach, e.g. proposed by the authors of [21], where an optimizer and a model solver work together to find optimal mappings specifically in the field of virtual machine technology. Computational complexity, which in most unrestricted cases is NP-hard, often prohibits application of centralized exhaustive methods above certain problem sizes, even as small as networks of only a handful of nodes. Other approaches try to restrict the solution space to tractable sizes by capturing important constraints,[24] but exhaustive search is still ineffective in practical problems, especially if we consider more than one QoS property of a configuration or more than one service at a time. Heuristics and approximative algorithms manage practical problem sizes more effectively. Malek et al. devised heuristic algorithms for software component deployment, based on greedy search and genetic programming in [27], approaching the problem from the user's perspective instead that of the service providers'. Besides, stochastic optimization appeared as an alternative first in [36], suggesting the use of the Cross-Entropy method as well.

Regeneration of replicas to remedy crashed components was proposed first by Pu [30] in the context of the Eden system. More recent systems, e.g. DARM,[29] provide automatic reconfiguration and regeneration of replicas in the context of group communication systems, and Om [41] focus on regeneration in a peer-to-peer wide-area storage system. Recently, with the advent of cloud computing, standardized cloud interfaces propose similar mechanisms for placement, migration and monitoring of components in the cloud.[13] Recent studies [4,19,21] show that the duration of virtual machine migration is in the order of 60-90 seconds. Service deployment support, that is intended to execute placement instructions given by our deployment logic are provided by such systems, however, focus is usually simply on failure recovery and improvement of availability without trying to optimize the new configurations and mappings. Authors in [40] show theoretically that replica placements of inter-correlated objects can impact system availability significantly if not chosen appropriately. Our work, on the other hand, focuses on improving both availability and overall performance.

Another centralized approach, namely group-finding algorithms to discover mappings in generic wide-area resource discovery is presented in [1]. In some way similar to the foraging behavior of our artificial ants, some approaches rely on extensive measurement data collection, however, our deployment logic does not store data centrally. Optimal placement of VMs under a variety of constraints has been focus of some research, e.g. in [35] and [21]. The SmartFrog[33]

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. Collections of components together with their configuration parameters can be activated and managed to deliver the desired services even in large-scale systems. The scale of these systems and the execution framework is close to the environment we envisage for the successful execution of autonomic component-based software services and which we target with our deployment logic. Configuration management in similar server environments based on fuzzy learning, targeting efficient resource utilization is investigated by Xu et al. in [39]. Biologically-inspired resource allocation algorithms appear in [17] to tackle service distribution problems. This is the path we too have chosen to follow while developing our deployment logic.

The basic method we built our deployment logic upon, the CEAS has been applied successfully to a variety of studies of different path management strategies, such as shared backup path protection, p-cycles, adaptive paths with stochastic routing, and resource search under QoS constraints.[15] Implementation issues and trade-offs, such as the management overhead imposed by additional traffic for management packets and recovery times are dealt with using a mechanism called elitism and self-tuned packet rate control. Additional reduction in the overhead is accomplished by pheromone sharing, where ants with overlapping requirements cooperate in finding solutions by (partially) sharing information (see [16] for details). In CEAS applied to routing, a routing path, from source to destination, is the target of the search. Instead of the cost of mappings, a routing path is evaluated in each iteration, i.e. a corresponding cost function is applied to the paths found. Furthermore, the pheromone values for routing CEAS is given by, $\tau_{ij,r}$ and represent an assignment between interface i and a node j at iteration r. Selection of the next hop for each ant, in this case, is based on the random proportional rule in the contrary to our algorithms.

§6 Conclusions

In this paper, we summarize our recent research towards obtaining an intelligent solution, a decentralized logic that is capable of finding near-optimal deployments for building blocks of services in a dynamic network environment. We presented how our bio-inspired heuristic approach looks for an efficient mapping between software components and nodes iteratively. Example scenarios were discussed ranging from the deployment of collaborating software components and management of replicas to virtualization in hybrid cloud environments. Additionally, an ILP model was shown that can be used to cross-validate the solutions found by our algorithm in an offline, centralized manner.

Many interesting paths of future work are considered. The algorithms presented are to be re-implemented in a scalable, Java-based simulator in order to explore larger scale scenarios. Increasing problem sizes are anticipated with the introduction of larger networks and, especially, larger amounts of parallel services. Nevertheless, efficient pheromone encodings provide the necessary reduction in the size of database structures and allow controlled increase in com-

plexity as problem sizes grow. Approximative methods, such as the heuristic algorithms presented in this paper, will continue to dominate on-line deployment decision making due to their flexibility and faster convergence. Besides, the core functions that describe the inner workings of the CEAS method, such as the pheromone or the temperature updates can also be revisited. The new, improved core functions in CEAS can possibly enhance the performance of the deployment logic.

The ILP model presented shall be extended to capture all the example scenarios. Regarding the scope of requirements, the next dimension to include is power-saving. Also, service models can be extended to capture additional aspects, such as consistency protocol costs. Besides, quantifying migration related costs is an interesting and difficult issue to look at in itself.

References

1) Albrecht, J., Oppenheimer, D., Vahdat, A. and Patterson, D. A., "Design and implementation trade-offs for wide-area resource discovery," *ACM Trans. on Internet Technology, 8, 4*, Sep. 2008.
2) Amazon Elastic Compute Cloud, Last checked: Aug 19, 2010. http://aws.amazon.com/ec2
3) Bastarrica, M. C., et al., "A Binary Integer Programming Model for Optimal Object Distribution," in *2nd Int'l. Conf. on Principles of Distributed Systems*, Amiens, Dec. 1998.
4) Clark, C. et al., "Live migration of virtual machines," in *2nd USENIX Symp. on Networked Systems Design and Implementation*, May 2005.
5) Csorba, M. J. and Heegaard, P. E., "Swarm intelligence heuristics for component deployment," in *16th Eunice Int'l Workshop and IFIP WG6.6 Workshop, LNCS 6164*, Trondheim, June 2010.
6) Csorba, M. J., Heegaard, P. E. and Herrmann, P., "Cost-efficient deployment of collaborating components," in *8th IFIP Int'l Conf. on Distributed Applications and Interoperable Systems*, Oslo, June 2008.
7) Csorba, M. J., Heegaard, P. E. and Herrmann, P., "Adaptable model-based component deployment guided by artificial ants," in *2nd Int'l Conf. on Autonomic Computing and Communication Systems*, Sep. 2008.
8) Csorba, M. J., Meling, H. and Heegaard, P. E., "Laying pheromone trails for balanced and dependable component mappings," in *4th Int'l Workshop on Self-Organizing Systems, LNCS 5918*, Zurich, Dec. 2009.
9) Csorba, M. J., Meling, H. and Heegaard, P. E., "Ant system for service deployment in private and public clouds," in *2nd Workshop on Bio-Inspired Algorithms for Distributed Systems*, Washington, DC, June 2010.
10) Csorba, M. J., Meling, H., Heegaard, P. E. and Herrmann, P., "Foraging for better deployment of replicated service components," in *9th Int'l Conf. on Distributed Applications and Interoperable Systems, LNCS 5523*, Lisbon, June 2009.
11) Dorigo, M., et al., "The Ant System: Optimization by a colony of cooperating agents," *IEEE Trans. on Systems, Man, and Cybernetics Part B: Cybernetics, 26, 1*, Feb. 1996.

12) Efe, K., "Heuristic models of task assignment scheduling in distributed systems," *Computer, 15, 6*, June 1982.

13) Elmroth, E. and Larsson, L., "Interfaces for placement, migration, and monitoring of virtual machines in federated clouds," in *8th Int'l Conf. on Grid and Cooperative Computing*, Lanzhou, Gansu, Aug. 2009.

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

15) Heegaard, P. E., Helvik, B. E. and Wittner, O. J., "The cross entropy ant system for network path management," *Telektronikk, 104, 01*, pp. 19–40, 2008.

16) Heegaard, P. E. and Wittner, O. J., "Overhead reduction in a distributed path management system," *Computer Networks, 54, 6*, pp. 1019–1041, 2010.

17) Heimfarth, T. and Janacik, P., "Ant based heuristic for os service distribution on adhoc networks," *Biologically Inspired Cooperative Computing*, 2006.

18) Helvik, B. E. and Wittner, O., "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.

19) Hirofuchi, T., Ogawa, H., Nakada, H., Itoh, S. and Sekiguchi, S., "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, Shanghai*, May 2009.

20) Hunt, G. C. and Scott, M. L., "The Coign Automatic Distributed Partitioning System," in *3rd USENIX Symp. on Operating Systems Design and Implementation*, New Orleans, Feb. 1999.

21) Joshi, K., Hiltunen, M. and Jung, G., "Performance Aware Regeneration in Virtualized Multitier Applications," in *DSN'09 Workshop on Proactive Failure Avoidance, Recovery and Maintenance*, Lisbon, Jun. 2009.

22) Karp, R. M. and Luby, M. and Marchetti-Spaccamela, A., "A probabilistic analysis of multidimensional bin packing problems," in *16th annual ACM Symp. on Theory of Computing*, Washington, DC, May 1984.

23) Kephart, J. O. and Das, R., "Achieving self-management via utility functions," *IEEE Internet Computing, 11*, pp. 40–48, 2007.

24) Kichkaylo, T. et al., "Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques," *Int'l. Parallel and Distributed Processing Symposium*, 2003.

25) Kraemer, F. A. and Herrmann, P., "Service specification by composition of collaborations - an example," in *Proc. of the 2006 Int'l Conf. on Web Intelligence and Intelligent Agent Technology, Hong Kong*, IEEE/WIC/ACM, 2006.

26) Kusber, R., Haseloff, S. and David, K., "An Approach to Autonomic Deployment Decision Making," in *3rd Int'l Workshop on Self-Organizing Systems, LNCS 5343*, December 2008.

27) Malek, S., "A User-Centric Framework for Improving a Distributed Software System's Deployment Architecture," *Proc. of the doctoral track at the 14th ACM SIGSOFT Symposium on Foundation of Software Engineering*, Portland, 2006.

28) Meling, H. and Gilje, J. L., "A Distributed Approach to Autonomous Fault Treatment in Spread," in *7th European Dependable Computing Conference*, IEEE CS, May 2008.

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

30) Pu, C., Noe, J. D. and Proudfoot, A., "Regeneration of replicated objects: A technique and its eden implementation," *IEEE Transactions on Software Engineering, 14, 7*, pp. 936–945, July 1989.

31) Rouvoy, R. and Beauvois, M. and Lozano, L., Lorenzo, J. and Eliassen, F., "MUSIC: an autonomous platform supporting self-adaptive mobile applications," in *1st workshop on Mobile middleware: embracing the personal communication device, Leuven*, Dec. 2008.

32) Rubinstein, R. Y., "The Cross-Entropy Method for Combinatorial and Continuous Optimization," *Methodology and Computing in Applied Probability, 1, 2*, 1999.

33) Sabharwal, R., "Grid infrastructure deployment using smartfrog technology," in *Int'l Conf. on Networking and Services, Santa Clara, USA*, pp. 73–79, Jul. 2006.

34) Stützle, T. and Dorigo, M., "A short convergence proof for a class of ant colony optimization algorithms," *IEEE Trans. Evolutionary Computation, 6, 4*, pp. 358–365, 2002.

35) Verma, A., Ahuja, P. and Neogi, A., "pmapper: power and migration cost aware application placement in virtualized systems," in *9th Int'l Conf. on Middleware*, pp. 243–264, Dec. 2008.

36) Widell, N. and Nyberg, C., "Cross Entropy based Module Allocation for Distributed Systems," in *IASTED Int'l Conf. on Parallel and Distributed Computing Systems, Cambridge*, Nov. 2004.

37) Wittner, O., "Emergent Behavior Based Implements for Distributed Network Management," Ph.D. thesis, NTNU, Dept. of Telematics, Norway, 2003.

38) Wood, T. and Shenoy, P. J. and Venkataramani, A. and Yousif, M. S., "Black-box and Gray-box Strategies for Virtual Machine Migration," in *4th USENIX Symp. on Networked Systems Design and Implementation, Cambridge, MA*, Apr. 2007.

39) Xu, J. et al., "On the use of fuzzy modeling in virtualized data center management," in *Int'l. Conf. on Autonomic Computing*, June 2007.

40) Yu, H. and Gibbons, P. B., "Optimal inter-object correlation when replicating for availability," *Distributed Computing, 21, 5*, pp. 367–384, Feb. 2009.

41) Yu, H. and Vahdat, A., "Consistent and automatic replica regeneration," *ACM Trans. on Storage, 1, 1*, pp. 3–37, Dec. 2004.

42) Zlochin, M. et al., "Model-based search for combinatorial optimization: A critical survey," *Annals of Operations Research, 131*, pp. 373–395, 2004.

Máté J. Csorba: He earned his M.Sc. degree in electrical engineering at the Budapest University of Technology and Economics (BME) in Budapest, Hungary in 2003. From 2001, he had been working with test systems at Ericsson's Test Competence Center in Budapest. He joined the Norwegian University of Science and Technology (NTNU) in 2007 as a research fellow, where he is working towards his Ph.D. degree. His current research interests include optimization, software architectures and service deployment in distributed systems.

Hein Meling, Ph.D.: He is an Associate Professor of Computer Science at the University of Stavanger, Norway. He received his Ph.D. in 2006 from the Norwegian University of Science and Technology. His research interests include byzantine fault tolerance, grid computing, distributed deployment configuration optimization and distributed event-based systems. He is the principle investigator on the IS-home and Tidal News projects funded by the Norwegian Research Council.

Poul E. Heegaard, Ph.D.: He received his Siv.ing. (M.S.E.E. in '89) and his Dr. Ing. (Ph.D. in '98) degrees from the University of Trondheim (now NTNU). Since 2006, he has been an Associate Professor at the Department of Telematics, Norwegian University of Science and Technology (NTNU), where he has been since 2009 the Head of Department. His research interests cover performance, dependability and survivability evaluation of communication systems.