Algorithms for Self-Organization and Adaptive Service Placement in Dynamic Distributed Systems

Artur Andrzejak, Sven Graupner,Vadim Kotov, Holger Trinks
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2002-259
September 17th , 2002*

E-mail: {artur_andrzejak, sven_graupner, vadim_kotov, holger_trinks} @hp.com

self-organizing algorithms, adaptive service placement, distributed systems, grid systems

In this paper we consider distributed computing systems which exhibit dynamism due to their scale or inherent design, e.g. inclusion of mobile components. Prominent examples are Grids - large networks where computing resources can transparently be shared and utilized for solving complex compute tasks.

One of the hard problems in this domain is the resource allocation problem and the related service placement problem. In this paper we discuss distributed and adaptive resource allocation algorithms performed in such dynamic systems. These algorithms assume that no global information about resource availability and service demand can be provided due to the scale and dynamism.

Interesting aspects of our approaches are the capabilities of self-organization and fault-tolerance. We analyze and "factor-out" these capabilities, making them also usable in the setting of other dynamic distributed systems, for example in mobile computing.

* Internal Accession Date Only
Approved for External Publication

Algorithms for Self-Organization and Adaptive Service Placement in Dynamic Distributed Systems

Artur Andrzejak, Sven Graupner, Vadim Kotov, Holger Trinks

Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 94304, USA

{artur_andrzejak, sven_graupner, vadim_kotov, holger_trinks}@hp.com

Abstract

In this paper we consider distributed computing systems which exhibit dynamism due to their scale or inherent design, e.g. inclusion of mobile components. Prominent examples are Grids - large networks where computing resources can transparently be shared and utilized for solving complex compute tasks.

One of the hard problems in this domain is the resource allocation problem and the related service placement problem. In this paper we discuss distributed and adaptive resource allocation algorithms performed in such dynamic systems. These algorithms assume that no global information about resource availability and service demand can be provided due to the scale and dynamism.

Interesting aspects of our approaches are the capabilities of self-organization and fault-tolerance. We analyze and "factor-out" these capabilities, making them also usable in the setting of other dynamic distributed systems, for example in mobile computing.

1 Introduction

Grid computing arose in the early 1990's in the supercomputing community with the goal of making underutilized computing resources easily available for complex computations across geographically distributed sites. The idea of the Grid is to provide a transparent and secure access to the resources by a software layer installed on the machines of participating organizations. This layer provides a multitude of functions, including resource virtualization, discovery and search for resources as well as the management of running applications. In addition to proprietary Grid software, two major software frameworks are in use today: the open-source Globus toolkit [26] and the Grid Engine [24].

A major development in Grids is the *Dynamic Grid Computing* [27]. This research trend focuses on harnessing *dynamic* resources in the Grid by providing the applications with self-awareness of their changing environment. For example, the applications will possess the capability to migrate from site to site during the execution depending on both the changing resource availabilities and their own needs. We envision this trend also as an answer to both the increasing scale of Grids and to the correlated high costs of their manual management. In this paper we anticipate this basic functionality and illustrate how it can be used to increase the degree of automation in Grid systems. Another trend in Grids stressing its dynamic nature is to integrate, develop and use services in a grid environment according to the *Open Grid Service Architecture* (OGSA) [8] (we will therefore use the terms *application* and *service* interchangeably in this paper).

Suitable placement of services or applications on resources is the primary factor for the economic utilization of underlying resources in such dynamic systems. A good solution for this problem prevents overloading server environments or the communication infrastructure, keeps resource utilization and response times in balance, and achieves higher availability and fault-tolerance. This paper describes and evaluates several algorithms which provide suitable service placement while considering fault-tolerance and self-organization.

As a by-product we study universal approaches and paradigms for controlling large and potentially instable distributed systems under the aspects of self-organization and fault-tolerance. We believe that the resulting insights are useful as building blocks for a multitude of related problems (e.g. resource revocation) in distributed systems with dynamic nature, such as those occurring in mobile computing and ubiquitous computing. These elements are partially independent of other aspects of the algorithms and can be "factored out" from the proposed approaches.

Overview of the paper. In Section 2 we discuss several issues related to management of dynamic distributed systems in more detail. We describe in more depth the problems and challenges in this field. We further illustrate the trade-off between algorithm reactiviness and

the solution quality. A part of Section 2 is devoted to defining functions that evaluate the placements of applications.

The first considered algorithm, based on the so-called Ant Colony Optimization, is presented in Section 3. This paradigm comes from the study of behavior of real ants and incorporates elements of machine learning via recording the best partial solution by a "pheromone". The approach has been applied successfully to a variety of problems, including routing in telecommunication networks, matching problems and the famous Traveling Salesman Problem. Its strengths are high scalability, the possibility of balancing solution time against solution accuracy and the robustness against failures of even large parts of the system.

In Section 4 we discuss an approach taken from the coordination of mobile robots, called the Broadcast of Local Eligibility (BLE). We extend this method to provide better scalability than the original solution and suggest improvements in terms of communication costs by applying gossiping algorithms. While this algorithm is simple and has short reaction time, the placement proposed by the algorithms might be far away from the optimum. Therefore the use of this algorithm is mainly for discharging of "hot-spots", less for optimizing service-to-server assignments.

The algorithm presented in Section 5 combines a notion of intelligent agents which represent groups of services with P2P-based overlay networks information services. The advantages of this novel approach are exploiting the self-organization properties of P2P-networks, high scalability and the ease of further extensions.

Section 6 discusses two simple algorithms, which are easy to implement yet do not let us expect a good placement quality.

In Section 7 we describe related work, while the Section 8 is devoted to the conclusion.

2 Management of Dynamic Distributed Systems

2.1 Problem Domain

Balancing demand and supply. A major aspect of grids is to match resource supply with application demand. Resource capacities should also be provided locally to where demands occur avoiding cross-network traffic. Since demands are fluctuating over time and locations, application placements need to be adjusted accordingly, ideally completely automated without human intervention. Such an automated service grid control system then transparently regulates service demands and supplies.

So far, most integrated management systems (in Grids and also other computing networks) are limited in regard to functioning in virtualized environments across organizational boundaries. Besides automated fail-over techniques in high-availability systems, management systems typically automate monitoring and information collection. Decisions are made by human operators interacting with the management system. Major service capacity adjustments imply manual involvement in hardware as well as in software. Systems need to be adjusted, re-installed and reconfigured, all expensive manual processes.

Centralized versus distributed management. The design of an automatic management system for Grids is closely related to the scale of the managed system and the rate of system changes. In an ideal case, all information about system state could be collected in a central instance, and as a consequence an optimal placement could be made (modulo the computational tractability of the problem). However, with increasing scale and rate of system changes, this solution becomes inappropriate. Another problem is fault tolerance.

Instead, we consider *distributed algorithms* for solving the placement problem. We further strengthen the scalability property by assuming that each individual distributed component of an algorithm has only partial information about the global state of the system. While this assumption leads to reduced communication and increased reactiviness, the obtained placement of services to resources cannot be expected to be optimal, i.e. only heuristic algorithms can work under these assumptions.

Dynamic Distributed Systems. Computational Grids and similar computational distributed systems are inherently dynamic due to their large-scale and complexity. Here by "dynamic" we mean the property of a frequently changing state of resource availability as well as the state of service requirements. In a system comprising 1000s of servers, changes such as server failure, overload or resource revocation might occur every few seconds. Similarly, resource demand will fluctuate in short time intervals.

These effects require adaptation of the system to new conditions on a permanent basis. While it could be possible to manage such a system by an army of human operators, this approach is certainly not economically viable and more error-prone. In our view, automatic management comes into place at this point. We believe that self-organization, fault-tolerance and adaptation to changes in supply and demand of resources are the key

elements to master this challenge on the top level, i.e. the application level.

Self-organization, fault-tolerance and adaptation. The term "self-organization" is not defined precisely in the literature. Intuitively, it describes an ability of a system to organize its components into a working framework without the need of external help or control. For our purposes we will understand *self-organization* as the capability of adding and removing system parts without the need for reconfiguration nor the need for human intervention. This aspect is of particular interest for us since (non-automatic) management of systems is an essential cost factor and source of a majority of errors.

The *fault tolerance* of a system is its ability to recover from transient and also possibly permanent failures without human intervention. There is a large amount of literature on fault-tolerant systems; however, it is mostly focused on fault tolerance of system components, and not on recovery of large and complex distributed systems. The interested reader is referred to [18].

Adaptation to changing demand/supply conditions is closely related to load balancing. Research on this topic has a long history in distributed systems. However, in most cases local ensembles of resources (such as multiprocessors or clusters of workstations) are considered, and stable "laboratory-like" conditions are assumed. In our case we have to meet a multitude of goals as discussed in Section 2.3; also, the large-scale and the dynamics of Grid-like systems make new approaches necessary.

Paradigms for mobile computing and ubiquitous computing. The challenges of dynamic Grid systems bear similarities to challenges of other highly dynamic (albeit smaller) distributed systems – those occurring in mobile computing or ubiquitous computing. We believe that many of the methods or techniques presented here can become applicable or can give rise to new paradigms in those areas. Additional motivation for this statement is the fact that the Grid is envisioned to comprise mobile computing devices, as stated in the OGSA roadmap [8].

Satyanarayanan points out in his paper [22] that in mobile systems the roles of a server and client become blurred at certain times, and mobile entities take both roles depending on the actual system conditions and resource supply. Such a scenario is closely related to a picture of "dynamic mini-Grids" with needs for constant adaptation of the computing loads. In this way, the approaches discussed in this paper become directly applicable.

To facilitate the application of the self-organizing elements and fault-tolerant properties in other domains, we discuss at end of most sections the "building blocks" for transfer of learned lessons and paradigms.

Basic assumptions. In the remainder of this paper, we assume some lower-lever system properties which are necessary for the functionality of the discussed algorithms. Specifically, we assume a basic mechanism which allows a server or other type of resource to join the system and notify its "neighbors" (e.g. resources in the same subnet) about its existence. Such mechanisms are provided in the lower network protocol layer, or by the resource discovery mechanisms in mobile systems. Note that we do not assume any central instance to be notified: informing only the neighbors is sufficient.

Another mechanism we build upon is the ability of each resource to measure its distance (in network hops or similar units) from other resources in the network. This ability enable building "maps" of other resources classified by their distance from a server. While this problem is not yet solved satisfactory, there are some promising approaches e.g. in the domain of P2P-systems [20].

2.2 Reactiviness and Solution Quality

One of the challenges of the service placement problem is to find algorithms that are both reactive and deliver high-quality solutions for the control scale we are dealing with. In practice, the responsiveness of an algorithm must be traded against the quality of a solution. Thus, responsiveness constitutes one parameter of the design space. Another parameter is the type of the control system, ranging from centralized to completely distributed. Since it is unrealistic to find one algorithm, which can be parameterized in both dimensions, we look at several approaches covering most of the design space.

Figure 1 summarizes tradeoffs for algorithms used for decision-making. The first chart symbolizes the dependency between the solution quality and time to find a solution. The second chart shows that centralized algorithms usually do not scale well compared to distributed algorithms. The next figure classifies four algorithms in regard to solution quality vs. reactiveness. Since being part of a control system, reactiveness of decisions is important. Reactiveness is understood as the time between detection an abnormality, for instance a sudden peak demand, and the final computation of a decision how the situation can be dealt with. Three time scales are considered: the "design" stage of an initial service placement, in longer periods reiterated as long-term adjustment process in the system; a mid-term period for periodic operational adjustments, and a shorter-term period for discharging sudden hot spots.

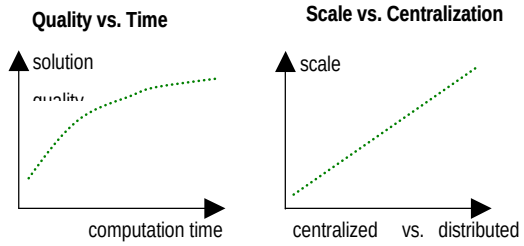

Figure 1: Decision-making algorithm tradeoffs.

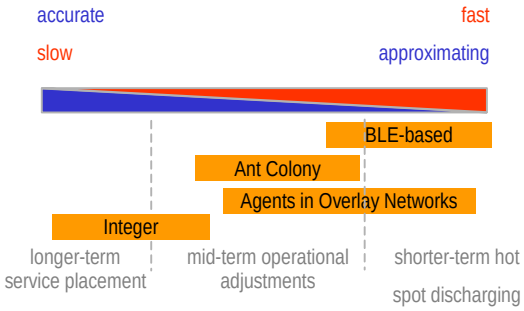

Figure 2: Comparison of four algorithms in terms of accuracy and reactiviness.

One approach we pursued is a centralized heuristic algorithm based on integer programming. This algorithm (not discussed in this paper) yields high-quality solutions but at a cost of longer running time and limited scalability. For improved responsiveness and larger scale, we explore agent-based and distributed algorithms described below. Such algorithms are composed of several simple decision-making instances, sometimes also referred to as agents. They communicate with each other directly or indirectly in order to approximate a solution. Each decision-making instance has, in general, only partial knowledge of the system. This facilitates scalability of such approaches. Furthermore, failure of any of the decision-making instance does not make the overall algorithm fail.

One agent-based approach is based on the Ant Colony Optimization paradigm [6], [23]. This fully distributed algorithm has medium responsiveness and can be used for periodical reassignments of services onto servers.

As an alternative approach, we evaluate an agent system based on a paradigm known as Broadcast of Local Eligibility (BLE), used for coordination of robot teams [28]. This partially distributed algorithm allows faster rebalancing of the managed services for the price of potentially lower-quality assignments.

Another approach uses more "intelligent" agents moving in the system guided by an self-organizing overlay network. This fully distributed approach can be parameterized in order to react either fast yet less optimal or slower but yielding a better-quality solution. It can be used for both fast discharging of hot spots or for mid-term operational adjustments. A comparison of these approaches in respect to the reactiviness/accuracy-tradeoff is presented in Figure 2.

2.3 Control Objectives and the Partial Objective Function (POF)

General control objectives. As discussed in the beginning of this section, the goals for optimal placement might vary in general. Therefore, the following algorithms are designed to be generic enough to support new objectives without fundamental changes. However, we focus on only few aspects to be achieved by control decisions. These are:

1. Balancing the server load such that the utilization of each server is in a desired range.
2. Placing services in such a way that communication demand among them does not exceed the capacity of the links between the hosting server environments.
3. Minimizing the overall network traffic aiming to place services with high traffic close to each other on nearby servers (nearby in the sense of a low number of communication hops across nodes).

The Partial Objective Function. We want to be able to compare different placement options in a quantitative way. To this aim we introduce a partial objective function (POF) f_{POF}, which is derived from a balanced sum of two characteristics. The first one, c_T, is the sum of traffic costs between the services on a pair of servers weighted by the distance of these servers. The second number, u_T, is the variance of the processing capacity usage among the servers. This leads to the POF computed by the formula:

$$f_{POF} = \frac{\beta}{\beta + (\alpha \cdot c_T + (1-\alpha) \cdot u_T)},$$

where α is the balancing factor between 0 and 1, and β a parameter described below. In our setting, both a lower weighted traffic cost and a lower variance are better. This is reflected in the value of the POF, which has a higher "score" for smaller c_T or u_T. Note that the value of f_{POF} ranges between 0 and 1; β must be chosen according to the maximum possible values of c_T and u_T in order to ensure a relatively uniform distributions of the values of the POF.

It is of course possible to exchange each of the above two characteristics. Especially, instead of c_T one might

imagine function which returns the total number of servers used for placement. Such a function can be implemented in the way described in [4].

Our POF is evaluated for a set V of servers and a set S of services. It is important to note that such a set might not contain all services or all servers in the system. In case of services this is motivated by a fact that for larger systems we can frequently isolate groups of interdependent services (i.e. services communicating with each other). While it makes sense to consider all services in such a service group for a placement decision, we do not need to consider services outside the group.

The rationale for considering only few and not all servers is dictated by scalability issues. In large systems, it is simply impossible to take all servers into consideration. The algorithms described in the following select an appropriate subset of the servers from the system in a heuristic fashion. The subsets are then subject to evaluation in the POF.

In the following, we give formal definitions for the characteristics c_T and u_T. We assume a fixed assignment of services in the set S to the servers in the set V.

For two servers v and v', we designate by $c_{v,v'}$ the estimated total traffic between all services placed on v and all services placed on v', measured in the number of exchanged IP packets. If $prox_{v,v'}$ is the network distance of servers v and v' (in terms of IP-hops), then the total weighted communication cost c_T is given by the formula:

$$c_T = \frac{1}{M} \sum_{v \in V} \sum_{v' \in V} prox_{v,v'} \cdot c_{v,v'},$$

where M is the total number of exchanged IP packets times the maximum distance between two servers in V.

For a server v, let u_v be the fraction of its processing capacity used by all services placed on this server. We assume that u_v is a real number in [0, 1]. Then the variance u_T of these numbers is defined by:

$$u_T = \sum_{v \in V} u_v^2 - \frac{1}{|V|} \left(\sum_{v \in V} u_v \right)^2 .$$

Necessary conditions of an assignment. An assignment must fulfill certain necessary conditions; for example, we cannot assign a service to a server with insufficient processing capacity. By slightly abusing the notion of an "objective function", we can use f_{POF} to ensure that such requirements are fulfilled. Specifically, we set the value of the POF to 0, if any of the following conditions is violated:

- Each service is placed at exactly one server.
- For each server v, the total processing demand of all services assigned to v is at most the processing capacity of v.
- For each server v, the total storage demand of all services assigned to v is at most the storage capacity of v.
- For each pair (v, v') of servers, the total network traffic between the services hosted on these servers should not exceed the link capacity between v and v'.
- The entries of a so-called affinity/repulsion matrix, if present, are respected; they indicate that a service must not or must be placed on a certain server.

3 Ant-Based Control Algorithm

In the classical Ant Colony Optimization [6], the path taken by an ant on its way between objects (e.g. cities in the Traveling Salesman Problem) represents a possible solution to the optimization problem. In our case, the objects would be both servers and services, and the alternating path would represent an assignment of services to servers. However, this approach is centralized and not really scalable for the following reasons:

1. The ant must "remember" the whole path it has taken; this information might become very large.
2. The ant must visit all objects on its tour. In a large and dynamic system, this is a serious drawback.
3. Finally, each solution (path) must be evaluated against others. This requires central knowledge.

3.1 Overview

For these reasons, we evaluate another approach not common in the classical Ant Colony Optimization yet leading to a better scalability. First we give an informal overview of this algorithm.

In our system, for each service *s* we instantiate a "demon" M_s called a *service manager* of s. If the service *s* is not yet placed or an overload condition has occurred, M_s creates multiple ants ("agents") and sends them out to the server network. Each ant has a *service list* containing s and the services cooperating with s. For each such a service, it knows the current resource requirements; also, it knows

the current communication requirements among the services in the service list.

The ant travels from one server to another choosing the servers along the path based on a probability computed locally. In each step, one of the services from the list is assigned to the current server. The path created in this way represents a partial solution to the placement problem as found by this particular ant. When the ant has assigned all the services, it reports its path to the service manager M_s of *s* and terminates. The manager compares the reported paths using the POF, where the set V of the POF is constituted by the servers visited by this ant, and the set S is the service list of s. This assignment is compared with the current placement of those services. Finally, M_s decides of a possible rearrangement of the placement.

On each server, the ant evaluates the score of the server in respect to each service from its list. For each pair (service, server), this placement score expresses how well this server is suitable to host the service. It is computed also using the POF in the way described below. Furthermore, the ant causes the pheromone table of the current server to be updated. This table contains *pheromone scores* for certain pairs (service, server). Those are essentially weighted sums of placement scores of the ants that evaluated this particular (service, server)-pair. The table is used to help an ant to decide which server to visit next. The server managers of neighboring servers periodically exchange these tables, thus providing a mechanism to disseminate the local information across the system.

3.2 Ants, Service Managers and Server Managers

In our algorithm we have three entities that store and manipulate data:

- a service manager M_s of a service s,
- an ant representing s,
- a *server manager* (corresponding to a single server) which executes the ant code, and maintains and updates the pheromone table of its server.

The data held by a service manager comprises the service list of s, the number of spawned ants and the currently best assignment reported by an ant. A service manager also knows how to evaluate the POF and its value for the current placement of the services in the service list.

An ant is launched with the following data that are "static" during its lifetime: the service list together with the current demand profiles of each service in the list, and the communication demand profiles between those services. This information is necessary to compute the score via a POF. The dynamic data carried by an ant are the scores of the already assigned services from the service list, and data about already visited servers, including link capacities.

Finally, a server manager holds the pheromone table of its server. The structure of the pheromone table is shown in Table 1. For each pair (serviceId, serverId) existing in this table, we record the known pheromone score, the age of this score and the number of ants which contributed to establish this score.

serviceId	serverId	pheromone score	score age (sec)	# ants
apache-01	15.1.64.5	0.572	95	15
apache-01	15.1.64.7	0.356	120	9
oracle-02	15.1.64.1	0.012	62	12
…	…	…	…	…

Table 1: An example pheromone table

3.3 Functionality of the System Components

In this section we describe in detail the behavior of the entities introduced above.

Service managers. A service manager constantly watches the performance of "its" service and evaluates the current assignment by a POF. On two occasions it spawns ants starting a process described below:

- If the POF value is larger than some critical limit; this corresponds to the case of an occurrence of a "hot spot".
- If a certain period of time has passed since the last launch of the ants. The purpose of this step is to periodically "rebalance" the wholes system towards an optimal utilization.

The process from the decision of launching ants until its termination includes the following steps.

1. Synthesize the ant data described in Section 3.2.
2. Place c_s copies of such an ant in the server network. The placement method and the value of c_s is described in Section 3.6.
3. Collect the assignments and the corresponding scores sent by the ants that terminated.
4. Once all ants have finished (or a timeout has occurred), compare the reported assignments by the POF and choose the one with the best POF value.
5. If the service s has already been placed, compare the current POF of s and the cooperating services with the one found in Step 4. If the new assignment is better by a threshold t_s

(representing the "penalty" for reassigning services to servers), continue with the next step; otherwise, terminate this epoch of ant launching.

6. If s is not placed, or the evaluation in Step 5. led to this step, reassign the services to servers in a following way.
    a. Contact all servers to be used in the new assignment of services and verify that their scores are still (approximately) valid. If this is not the case, start a new epoch of ant launching (i.e. begin from Step 1.)
    b. Contact the service managers of all cooperating services and let them stop any running ant-based evaluations.
    c. Contact the servers to be used in the new assignment and let them reserve the required resource capacities.
    d. Start installing and starting the services on their new locations.
    e. When step d. is finished, shut down the services in the old placement.
    f. Finally, start the service managers of the newly installed services.

Ants. An ant created by a service manager M_s of a service *s* "travels" from one server manager to the next one (usually residing on an another physical entity). Technically, it is done by contacting the next server manager, transmitting the ant data to it and initiating executing the ant code for this ant instance. The choice of the next server manager is done in the way described below. The ant has the following life cycle after it has arrived on a new server manager:

1. Evaluate for each service in the service list the score in regard to this server. This is done via the POF for this server as described in Section 3.4.
2. Update the pheromone table of the current server by passing the computed scores to the server manager.
3. Choose the service with the highest computed score among the not yet assigned services and remember this assignment.
4. If all services from the internal list have been assigned, report the resulting assignment to the "original" service manager M_s, then terminate.
5. Otherwise, move to the next server manager and continue with 1.

Server managers. Both entities described above have essentially a fixed order of tasks to be executed. By way of contrast, a service manager acts in an asynchronous way, providing "services" to the other two entities. Its roles comprise the following tasks:

1. It provides an environment where the ants are executed. Especially, it can asynchronously receive messages from other server managers which send the ant data. Once this data is received, it executes the locally stored code representing an ant.
2. It lets an ant update the pheromone table with the scores computed for the services in the service list.
3. It maintains the pheromone table by updating the age of the pheromone scores and pruning the table. The last step is necessary, because in the extreme case, the pheromone table could attain a size proportional the to number of servers multiplied by the number of services; this would seriously impede scalability. During the pruning, the oldest entries (except for those regarding the neighboring servers) are removed, until the desired table length is reached.
4. Finally, a server manager sends periodically its own pheromone table to the neighboring servers, keeping the information of the neighbors up to date.

The last function provides a mechanism for dissemination of the local knowledge throughout the system. This reduces the gap between a distributed system where each participant has only local knowledge, and a centralized system with the complete, global knowledge of the system. The size of the time interval between the updates and the size of a pheromone table controls the degree of the "global knowledge" in the system. An antagonistic trend is the rate of changes in the system and consequently the ageing rate of the pheromone. Also, albeit a high degree of this knowledge is very useful for choosing the next server in a correct way, attaining it costs a lot of resources, mostly network bandwidth and storage for the pheromone tables.

3.4 Placement Scores and the Pheromone Table

Recall when an ant reaches a new server manager, it computes placement scores for all services from its list in respect to the current server. For such a pair (service s, server v), this computation is done via the POF as follows. The current server v and servers already visited by an ant become the set V. Furthermore, s and all already assigned services from the ant's service list

constitute the set S. Then the value of the POF is computed for the (partial) assignment of services to servers already chosen by this ant, together with the mapping of s to v. We assume that information about link capacities between the servers is buffered by the ant or can be obtained from the server manager, if necessary.

Let us describe now how the pheromone tables are updated. Assume that an ant has computed a fresh placement score r for the pair (service, server). If such a pair does not exist in this server manager's table, it is simply inserted with the pheromone score being equal to the placement score. Otherwise, the new value p' for this pair's pheromone score is computed from the current table entry p and the newly computed placement score r by the formula:

$$p' = \gamma \cdot p + (1-\gamma) \cdot r.$$

Here γ is a parameter between zero and one which determines the degree of inheriting previous pheromone score value. Note that the contribution of all other scores decreases geometrically with the number of iterations: if the very first ant which has visited the node has set the pheromone score to p, then after k new ants have reached the server, the contribution of this first ant to the current score of the pair will only be $\gamma^k p$.

We also want to consider an effect known from the ant colony systems in the nature: evaporation of the pheromone. Due to this effect, old and probably outdated information about affinities of services to servers will be removed with time, even if no new ants have arrived at this server. To this aim, a server manager scans through its pheromone table once every T minutes, and reduces the score p in the pheromone table according to the formula:

$$p = \delta \cdot p,$$

where delta is an aging factor between 0 and 1 (usually close to 1). If the value of the pheromone score decreases below a certain limit, these pairs are removed from the pheromone table in order to save storage resources.

3.5 Choosing the Next Server

Pheromone tables are the main decision factor for choosing the next server to be visited by an ant. Since those tables are exchanged by the neighboring servers and propagated through the system, an ant has a good chance to find a pair (service s, server v) in the pheromone table of the current server. Here v is a not too distant server and s is a still unassigned service from the service list of this ant. If multiple such pairs have been found, the server of a pair with the highest pheromone score is selected. However, if no such a pair exists, the ant chooses a set of servers from the pheromone table with 1. most recently updated pheromone scores, and with 2. highest pheromone scores. Then a random server from such a set is selected as the next host. This approach targets to identify with high probability servers with free computational resources.

As an alternative to each of the above cases, sometimes we send an ant to a randomly selected not-too-distant server. The decision for this step is taken with a (small) probability h. Such an addition of a "noise" is helpful to prevent the *blocking problem* and the *shortcut problem* [23]. The blocking problem occurs if a "popular" path found by many ants can no longer be taken, e.g. due to a server failure. The shortcut problem occurs in a situation where a new assignment of services to servers suddenly becomes possible, for example due to introduction of new servers to the system. In both cases the information stored in the pheromone tables might cause lack of adaptation of the ants to the new conditions. A small amount of noise forces the ants to exploit the alternative routes on a permanent basis.

3.6 Initial Placement of the Ants

The initial placement of the ants is intuitively an important factor for finding good service placements. In our case, the service manager M_s places the ants in the system according to the following schema.

First, it determines N_r "regions" where clusters of ants are placed. The centers of these regions are chosen randomly in the known system area in the way that the probability of choosing a center distant from the service manager is smaller than choosing a center close to M_s. To this aim, each service manager maintains a (partial) map of the resources in term of their network location. The resources are categorized by their IP-distance d to the server manager. When choosing a center of the region, in the fist step the service manager selects randomly a class of resources with a distance d to M_s. Then it decides to continue with this class with probability

$$\frac{1}{(1+d)^{\theta}},$$

otherwise it chooses again a random class until success; here θ is a parameter greater 1. If successful, a random resource as the center of a new region is chosen. According to the findings in [15], this approach ensures that very rare resources can be still discovered, but simultaneously supports clustering of services according to the location of their inception.

In each of the regions determined in this way, the service manager spawns N_a ants on the resources close to the

center of the region. Here a similar approach to the one described above is taken, yet the distances of the created ants from the center of the region are kept smaller by means of increasing θ. Furthermore, ants "repel" themselves: if an ant is placed on a certain resource, then M_s will discard all servers within a distance D_r from this resource for further placements.

3.7 Conclusions for Self-Organization and Fault Tolerance

The presented algorithms have some pleasant features in respect to automating resource management. For example, servers and resources added to the network do not need to inform any central instance of their existence; it is sufficient, that only their neighbors learn new topology (by the mechanism mentioned in Section 2.1). Furthermore, even if the majority of the servers in the system are unavailable or unreachable, our approach will not be prevented to work correctly in the remaining part of the system. Also a plus is the fact that by changing the amount of noise in the ant's selection of its next steps along its path we can adjust the necessary degree of adaptability.

A disadvantage is the fact that the service manager is a single point of failure; if it disappear, the service or a group of them might not recover without human intervention. The reader is referred to Section 5 for a solution to this problem.

Building blocks for other domains. We believe that the idea of pheromone tables deserves some attention in conjunction with the agent technology. In the classical agent frameworks, the communication takes place either directly between agents or between agents and "agent containers" (i.e. the environment executing them). It would be interesting to exploit models where the information between agents can be exchanged in an undirected and passive ways, as in the case of pheromone tables. However, we are not aware of possible applications of this schema.

Another idea worth to be "extracted" from the above algorithm is the dissemination of information by exchanging it between the neighboring servers only. This mechanism, similar to those used in Systolic Computing, allows to blur the distinction between a situation where only partial information of the system is known at each node as opposed to the scenario in which every node possess a complete system description. It would be interesting to learn by theoretical analysis or an empirical study how frequently information must be exchanged and how fast the information can expire for a large part of the system to have accurate information.

4 BLE-Based Control Algorithm

We adapt the concept of the Broadcast of Local Eligibility used for coordination of robots [28] for the placement of services. This concept can be used to create highly fault-tolerant and flexible frameworks for coordination of systems of agents. However, the originally proposed framework has a drawback of limited scalability. To overcome this problem, we use a hierarchical control structure discussed below.

Decision cycle in a cluster. We consider a cluster of servers with a distinguished server called *cluster head*. Each member of the cluster has the ability to broadcast a message to all other members of the cluster. This can be done either directly or via the cluster head. The placement of services in this cluster is periodically re-evaluated by arbitration between peer servers in so-called *decision cycles*. The time between two cycles is determined by the required responsiveness to fluctuations in server utilization and by the induced communication between cluster members.

In each decision cycle, the following actions take place:

1. Each server broadcasts the list of services it hosts with all new arrived services and simultaneously updates its list of all services in the cluster.
2. Each server evaluates its own suitability to host each service and sorts the list according to the computed score. The evaluation is done by using the POF from Section 2.3. In addition, a service already deployed on a server highly increases the score.
3. Each server broadcasts a list, ordered by scores, of those services the server can host simultaneously without exceeding its capacity.
4. When a server receives a score list from a peer, it compares this score with its own score for a service. As a consequence, each server knows whether it is the most eligible one for hosting a particular service.
5. The changes in the service placement are executed. Notice that each server knows already whether it has to install new or remove current services. In addition, the cluster head compares the initial list of services with those, which will be hosted at the end of this decision cycle. The remaining services are passed on to the next hierarchy level as explained below.

An important aspect is that the servers do not forget the list of services in the cluster after a decision cycle. In this way we provide fault-tolerance: if a server hosting certain services fails, other servers in the cluster will automatically install the failed services (or the cluster head adds them to the list of unassigned services).

Gossiping algorithms. Note that steps 1 and 3 require *all-to-all* communication, i.e. each server learns the information from all other servers. This may lead to a problem of the communication costs in terms of the number of messages and the time until all members of a cluster are informed. In infrastructures like Ethernet or wireless LAN a cost of a broadcast is comparable to sending a targeted message, which partially relieves the situation. This problem becomes more serious if members of a cluster are geographically distributed or communicate over a switched network.

These communication costs can be reduced using *gossiping algorithms* [10]. These deterministic and also randomized [14] algorithms achieve optimal bounds for the number of messages with a low number of communication rounds; for example, the information exchange can be completed in approximately $2 \log_2 n$ steps in the deterministic case, and in roughly $\log n$ steps in the randomized case, where n is the number of servers in the cluster. The reader is refereed to the literature for more detailed discussion.

Scalability by a cluster hierarchy. Obviously, the scalability of the above approach is limited by the size of the cluster, the communication capacity in the cluster and the processing capacity of the cluster head.

We propose a following hierarchical approach to extend the scalability. Basically, the cluster heads of the clusters at level k are treated as "normal" members of a cluster of level k+1. However, they compete only for those services, which could not be installed in their own cluster (see step 5. above). After a decision round in the cluster of level k+1, these pending services are possibly moved to another peer, which is a cluster head for a cluster of level k. (The cluster head evaluates the eligibility of the servers in its own cluster, not its own eligibility). In the cluster of level k, these services become part of the list of services to be installed and participate in the normal decision cycles.

The cluster size is essential for the balance between the responsiveness of the system and flexibility. Identifying a correct hierarchical structure can be done similarly to clustering algorithms used in sensor networks [7].

4.1 Conclusion: Self-Organization and Fault-Tolerance

The above algorithm has several good properties. In addition to being relatively simple, it ensures the automatic recovery of services without special mechanisms. Also, the size of a cluster can be treated as a parameter for tuning the algorithm's reactiviness (against solution quality); see Section 2.2. A weakness of the algorithm is the fact that the cluster head can become overloaded or even temporarily be a single point of failure (however, cluster heads building the cluster of the next level will recover the failed head in their next decision cycle). Another inconvenience of this algorithm is the fact that the hierarchy of clusters must be created externally (i.e. is not given implicitly by the algorithm), which limits the self-organization of this approach.

Building blocks for fault-tolerance. An interesting quality of the above approach is the implicit fault tolerance and also implicit "negotiation" between the resources about their assumed roles (i.e. roles as hosts for applications). This mechanism works due to the fact that information about all required tasks (in our settings, the services to be hosted) and information about the capabilities of the cluster members is known to everybody in a cluster. While this scheme has been exploited successfully in the BLE-approach, we think that by extending it to a hierarchical system of cluster a true scalability becomes possible.

5 Agents in Overlay Networks

In this section we describe an approach which combines the advantages of agent technology techniques with the fault-tolerant properties of peer-to-peer (P2P) networks.

Service groups and agents. As discussed in Section 2.3, services frequently build clusters of interdependent entities, which do not rely on further services outside the cluster. Such a *service group*, if not too large, can be treated as one (albeit not atomic) entity in the process of the optimization. Therefore we assign to such a service group N_a instances of *group agents*. Each group agent has the task to walk around in the resource network and evaluate the current server and its neighborhood in regard to placement of the services in the service group; however, one agent stays on one of the servers which host members of the service group, and evaluates only the current placement.

The evaluation of potential new placements is initiated by retrieving the capacity parameters and utilization data of the current server and its neighboring servers by means of a P2P-network described below. This data is then a subject to evaluation by the Partial Objective Function from Section 2.3. Periodically, the group agents belonging to the same service group exchange their best scores. If the score of one of them is better than the real placement (also taking into account a penalty for moving services), this group agent initiates a rearrangement of the placement.

A further assignment of a group agent is to provide the fault-tolerance to the optimization infrastructure: it is done by constantly watching all other N_a-1 group agents for being alive; the special group agent staying close to

actually deployed services also watches the health of the services. If one of the group agents fails, it is immediately re-instantiated by other agents. Also, if one of the services turns out to have failed, an appropriate recovery action is initiated.

It is important to note the difference to the Ant Colony Optimization Algorithm presented in Section 3. While both ants and agents use a notion of a service group and carry data of services in such a group, agents have a different evaluation algorithm compared to ants. While an ant assigns a service to a server in each step, an agent evaluates a possible assignment of *all* services to the current server and its neighbors in such a step. Furthermore, agents have more "intelligence" and do not die as opposed to ants. On the other hand, ants use the pheromone trails to learn the best assignments.

P2P-based overlay networks. Since the evaluation of a new agent placement incurs a lot of effort, the next jump of an agent must be chosen carefully. To this aim agents are guided by information from an overlay network which provides capacity-related attributes of servers. In the overlay network described in [3] servers are connected in a P2P-manner to achieve fault-tolerance and self-organizing properties (i.e. servers may join and leave without a reconfiguration exercise). The functionality of the network allows range queries of attributes; in our case we are mostly interested in server processing capacity, server storage capacity and the *density* values of these attributes. The density of an attribute is the averaged attribute value from a group of servers whose center is the server which "labels" this density value; thus, a density value is an indicator of the attribute (capacity) in the surrounding of a server. The density values are periodically computed on each server by receiving updates from the surrounding resources.

When deciding about the next server to be visited, an agent first collects the current utilization data from its service group. This demand value determines the range for which the density values are queried. The overlay network responds with a list of servers fulfilling the criteria. An agent sorts them according to their distance, and chooses randomly the next server to move on, similarly as described in Section 3.5. Once arrived on the new server, it queries directly the surrounding servers retrieving their individual attribute values. (If ranges of values are necessary, the overlay network query capability can be used.). This data is then used for the evaluation of the POF.

5.1 Lessons Learned for Self-Organization and Fault-Tolerance

Opposed to the ACO-approach from Section 3, the above algorithms provides full fault-tolerance. Since agents are guarding themselves together with the service group, faults of even a majority of the system does not lead to breakdown of the service group. Another positive aspect is exploiting the self-organization properties of the underlying P2P-network. A disadvantage of the algorithm is the fact that each agent is a complex entity, which might bind more resources than e.g. in case of the Ant Colony Optimization-based algorithm.

Building blocks. The first idea deserving to be transferred into other research domains is the symmetry of agents in their roles (except for the one which stays at the service group). This simplifies the overall schema and allows a higher degree of fault tolerance. Another noteworthy paradigm is using a P2P-network as a "lower layer" providing the self-organization capabilities to the system elements (and in our case, also providing the information infrastructure). Such an architecture suggests a layered model, where lower layers provide self-organizing properties used by higher, more complex layers.

6 Two Simple Algorithms

To complement the above three approaches, we discuss in the following two algorithms for service placement characterized by simplicity and statelessness.

6.1 Random / Round Robin (R3) Load Distribution Algorithm

Pretty much like random or round robin scheduling, the load distribution algorithm pushes load from an overloaded server to a randomly or in a round robin fashion chosen neighbor that may absorb that load if it has the capacity, or pushes the load further on to another server chosen in the same fashion. Once a place has been found where the load can be absorbed, the actual migration of the load is initiated in the underlying system.

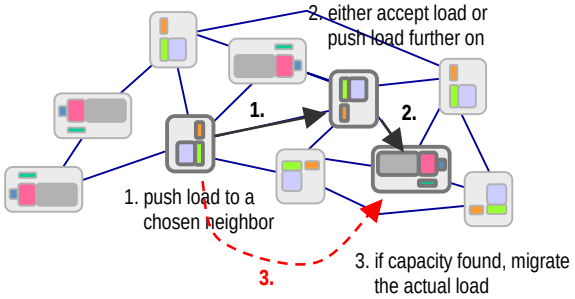

Figure 3: The R3 Load Distribution Algorithm.

The advantage of this algorithm is its simplicity and statelessness (efforts to maintain states can be avoided). The disadvantages are unpredictability and insufficient (random) convergence on the chance for thrashing.

The termination problem of the algorithm can be addressed by limiting the number of hops. Cycles cannot be avoided due to the statelessness of the algorithm.

6.2 Simple Greedy Algorithm

Greedy Algorithms also represent a simple category of distributed algorithms. A simple greedy algorithm just pushes load on to the least loaded neighbor. Unlike random algorithms that do not take any information into account, greedy algorithms make use of locally available information such as load conditions on neighbored servers in our case. Servers need to exchange information in order to keep this information up to date. However, total consistency cannot be achieved.

Convergence is better than random. However, since load is pushed only to most-underutilized servers, these servers quickly become utilized with the danger of becoming overloaded themselves. This causes greedy algorithms tend to oscillate with the effect of thrashing service load in the underlying system. For this reason, greedy algorithms strongly depend on the update frequency of load conditions in the meta-system. They also require a bias between load states in order to defuse the oscillation problem. Termination and cycles can also not be avoided by the algorithm itself. Both need to be guaranteed by limiting the number of hops. The algorithm does not guarantee to find a solution.

The algorithms R3 and Greedy make good use of locality by placing load on the closest server they can find. Over a longer period, both algorithms achieve good load balancing. However, fast reactiveness is not guaranteed.

7 Related Work

Work related to the topics of this paper can be classified in three themes: self-organization of distributed systems; resource management in such systems; and distributed constraint solving.

The self-organization of distributed systems includes contributions from P2P-systems research, mobile systems and ubiquitous computing. In most P2P-systems mechanisms which automatically handle joining and leaving nodes (e.g. servers) are inherent parts of the design. Examples include Gnutella, Pastry, Tapestry, Chord and CAN **Error! Reference source not found.**. Project OceanStore [16] exemplifies an application of a P2P-based self-organization for resource management; another example is given in [3].

The concept of ad-hoc networks known from mobile computing [21] is another source of paradigms for self-organization. The focus of the research in this area are protocols for discovering routes between dynamically located nodes. The BARWAN project [5] addresses aspects of self-organization and dynamic adaptation in the domains of mobile networking and ubiquitous computing.

The most prominent project at the edge of self-organization and resource management is IBM's Autonomic Computing vision [13]. This broad collection of projects intends to create systems that are self-configuring, self-healing and self-optimizing. Related to this research thread is the Océano project [12]. It addresses the designing and building a prototype of a scalable infrastructure for a large-scale "computing utility powerplant" that enables multi-customer hosting on a virtualized collection of hardware resources. An undertaking of similar flavor is HP's Utility Data Center project [11], [1].

There is a multitude of activities focused on using computational Grids for sharing distributed supercomputing resources [25], [2]. Examples include the Globus toolkit [26], or Sun's Grid Engine [24]. Although these systems exhibit a mature infrastructure for resource management, the scheduling part still lack more sophisticated algorithms.

In the field of distributed constraint solving the most notable thread is the research on the Distributed Constraint Satisfaction Problems (DCSPs) [17]. In a DCSP several computational agents try to solve a connected Constraint Satisfaction Problem collectively. Such a problem consists of a set of variables which take their values in particular domains, and a set of constraints which specify the permitted value combinations. Each agent carries – strategy dependent – a subset of variables or a subset of values for variables and tries to assign values to variables while preserving consistency between agents. Noteworthy strategies in DCSP are Asynchronous Backtracking, Weak-Commitment Search Algorithms or Distributed Constrained Heuristic Search [9].

8 Conclusion

The algorithms presented in this paper provide means for distributed control of resources dynamic distributed systems such as large Grids or federations of data centers. The approaches exhibit different levels of the tradeoff between reactiviness and solution accuracy, so that not a single algorithm but a suite of them becomes necessary. Interesting aspects of the algorithms are the capabilities of self-organization and fault-tolerance. For each algorithm, we discuss these capabilities with the goal of proposing

paradigms usable in other domains, such as mobile computing or ubiquitous computing.

Figure **4** summarizes and classifies the behavior of algorithms (a comparison to a centralized integer programming approach not discussed in this paper is also provided).

	Integer	Ovl.Agts	Ants	BLE	R³	Greedy
scalable	-	+	+	+	+	+
dense graph	+	-	+	-	?	?
globally accurate	+	-	?	-	-	-
fast reactivenes	-	+	-	+	-	-
self-organi-sation	-	+	+	+	?	?
fail-over capability	-	+	-	+	?	?
extensib-ility	+	+	-	-	+	+
adaptability	-	+	+	-	+	+
simplicity	+	-	-	+	+	+

Figure 4: Classification of control algorithms.

References

[1] A. Andrzejak, S. Graupner, V. Kotov and H. Trinks: *Control Architecture for Service Grids in a Federation of Utility Data Centers*, HP Labs Technical Report[1] HPL-2002-235, 2002.

[2] A. Andrzejak, S. Graupner, V. Kotov and H. Trinks: *Self-Organizing Control in Planetary-Scale Computing*, IEEE International Symposium on Cluster Computing and the Grid (CCGrid), May 21-24, 2002, Berlin.

[3] A. Andrzejak and Z. Xu: *Scalable, Efficient Range Queries for Grid Information Services*, Second IEEE International Conference on Peer-to-Peer Computing (P2P2002), Linköping, Sweden, 5-7 September 2002.

[4] A. Andrzejak, J. Rolia, and M. Arlitt: *Bounding the Resource Savings of Several Utility Computing Models for a Data Center*, in preparation, 2002.

[5] E. A. Brewer, R. H. Katz, E. Amir, H. Balakrishnan, Y. Chawathe, A. Fox, S. D. Gribble, T. Hodes, G. Nguyen, V. N. Padmanabhan, M. Stemm, S. Seshan and T. Henderson: *A Network Architecture for Heterogeneous Mobile Computing*, IEEE Personal Communications Magazine, Oct. 1998.

[6] M. Dorigo, V. Maniezzo and A. Colorni: *The Ant System: Optimization by a Colony of Cooperating Agents.* IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26(1), 29-41, 1996.

[7] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar: *Next century challenges: Scalable coordination in sensor networks*, Proceedings of MOBICOM, pp. 263-270, Seattle, USA, August 1999.

[8] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke: *The Physiology of the Grid – An Open Grid Services Architecture for Distributed Systems Integration*, DRAFT, http://www.globus.org/research/papers/ogsa.pdf, May 2002.

[9] M. Hannebauer, On Proving Properties of Concurrent Algorithms for Distributed CSPs, ** complete **

[10] S. T. Hedetniemi, S. M. Hedetniemi, and A. L. Liestman: *A survey of broadcasting and gossiping in communication networks*. Networks 18: 319-349, 1988.

[11] HP, *Utility Data Center*, http://www.hp.com/go/hpudc, http://www.hp.com/go/always-on, November 2001.

[12] IBM, and University of Berkeley, *Oceano Project*, http://www.research.ibm.com/oceanoproject.

[13] IBM, *Autonomic Computing*, Manifesto, http://www.research.ibm.com/autonomic/manifesto.

[14] A.-M. Kermarrec, L. Massoulie, and A. J. Ganesh: *Reliable Probabilistic Communication in Large-Scale Information Dissemination Systems*, Microsoft Research Technical Report MMSR-TR-2000-105, October 2000.

[15] J. Kleinberg: *The Small-World Phenomenon: An Algorithmic Perspective*, Cornell Computer Science Technical Report 99-1776, October 1999.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao: *OceanStore: An Architecture for Global-Scale Persistent Storage,* ASPLOS '00, MA, USA, 2000.

[17] Q. Y. Luo, P. G. Hendry, and J. T. Buchanan: *Comparison of different approaches for solving distributed constraint satisfaction problems*, Research Report RR-93-74, Department of Computer Science, University of Strathclyde, Glasgow G11XH, UK, 1993.

[18] E. Marcus and H. Stern: *Blueprints for High Availability: Designing Resilent Distributed Systems*, John Wiley & Sons, N.Y., 2000.

[19] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja1, J. Pruyne, B. Richard, S. Rollins and Z. Xu, *Peer-to-Peer*

[1] HPL-TR are available: http://lib.hpl.hp.com/techpubs.

Computing, HP Labs Technical Report HPL-2002-57, 2002.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker: *A Scalable Content-Addressable Network*, SIGCOMM 2001, San Diego, August 27-31, 2001.

[21] E. M. Royer, C.-K. Toh: *A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks*, IEEE Personal Communications Magazine, Apr. 1999.

[22] M. Satyanarayanan: *Fundamental Challenges in Mobile Computing*, Symposium on Principles of Distributed Computing, 1996.

[23] R. Schoonderwoerd, O. Holland, J. Bruten, and L. Rothkrantz: *Ants for Load Balancing in Telecommunications Networks*, Adaptive Behavior 2:169-207, 1996.

[24] Sun Microsystems, *The Sun Grid Engine*, http://wwws.sun.com/gridware.

[25] The Global Grid Forum, http://www.gridforum.org/.

[26] The Globus Toolkit, http://www.globus.org/toolkit.

[27] The GridLab Project, http://www.gridlab.org

[28] B. B. Werger, and M. Matarić: *From Insect to Internet: Situated Control for Networked Robot Teams*, to appear in Annals of Mathematics and Artificial Intelligence, 2000.