A pheromone-based coordination mechanism applied in P2P

Kurt Schelfthout[1] and Tom Holvoet[1]

KULeuven, Department of Computer Science, Celestijnenlaan 200A,
3001 Leuven, Belgium
{Kurt.Schelfthout, Tom.Holvoet}@cs.kuleuven.ac.be
http://www.cs.kuleuven.ac.be

Abstract. In this paper, we discuss the principle of synthetic pheromones, which we view as a high level coordination mechanism suitable for highly scalable, open and self-organized systems, of which peer to peer systems are an example. We present a stable abstraction for the application of synthetic pheromones, building on an existing coordination mechanism, objectspaces. The coordination principle is evaluated on the problem of search in a file-sharing P2P system.

1 Introduction

Some applications, notably distributed applications, have very stringent requirements concerning decentralization, scalability and robustness. For example, a peer to peer system needs to deal with a very large number of peers, joining or leaving the system at will, preferably operating without a central component. Other examples include active networks [10] [6](networks that are able to dynamically reconfigure themselves in response to faults or changing quality of service requirements); manufacturing control [11](where it is difficult to gather and plan centrally because information may be outdated very rapidly); autonomic computing [8] (the promise of self-healing, self reconfigurable systems).

We distinguish three characteristics of these applications:

Scale-free: The attribute scale has two dimensions: space and time. We call a distributed system scalable in space when infinitely many hosts or peers can be added to the system. Scalability in time means to be able to cope with distributed sources of information, and being able to process the information in a distributed way. This is necessary when the sheer amount of information is too big to gather and process on a central server, for example due to time constraints. It is clear that a far-reaching decentralization of the system will be a consequence of this requirement - any central component could ultimately prove a bottleneck for the scalability of the system.

Openness: An open system, in our view, means that any entity may leave at will without breaking the system (it may however influence the performance in some way), or that new entities may join the system at will (Note that this is another definition as for example open in “open source” - we make

no claims concerning the possible heterogeneity of the system). This characteristic says something about fault-tolerance - it says that no entity, when failing, should bring the entire system down with it. In such a system, a fault is not an exception, but rather the possibility of error must be taken into account every step along the way when designing.

Self-organized: Since such a system cannot rely on any outside source to be organized by, because this would break the scale-free and openness requirements, it must by definition organize itself.

Our focus is on the aspect of coordination of such a system: given the inherent distributed nature, we investigate what mechanisms can be used to allow for a coherent behavior of the whole system. We will first discuss our approach to this problem in section 2, and give a short introduction to the concept of objectspaces. We then introduce synthetic pheromones in section 3, explaining our implementation vehicle for pheromones, called dynamic objects. Then the approach is validated on a file-search problem in a peer to peer network. After discussing related work, we reach the conclusion.

2 Approach

Multi-agent systems (hereafter written MAS) seem like a natural way of tackling the problems associated with the class of applications described above. An agent is then seen as a problem solving entity, that is *situated* in a local environment (i.e. is no global or centralized component and has no complete view of the system), and is naturally able to adapt itself to harsh circumstances. However, the designers of MAS for this (ever-growing, considering the Internet's growth) class of applications could benefit greatly from a *coordination infrastructure* that abstracts certain patterns of coordination between agents. Indeed, not only the agents themselves need to be adaptable, but also the coordination mechanisms they use to reach a certain goal need to be appropriate. For example, it seems unreasonable to use negotiation protocols in the aforementioned settings, since so many exceptional situations can arise that this will place too large a burden on the application programmer. Instead, we focus on coordination mechanisms where no central component is needed (for scalability reasons), and that can operate efficiently even in the face of failures (for openness reasons). By using appropriate coordination mechanisms, the agents can be said to self-organize, which in turn leads to a very robust system.

The ultimate goal is the development of a coordination infrastructure for multi-agent systems. A coordination infrastructure provides mechanisms by which agents can coordinate their activities in a concurrent and distributed world. A simple example of coordination would be "waiting for a result". The main focus is on achieving coordination through environment-mediated communication - this means that there is no *direct* interaction between agents, instead agents communicate or interact *indirectly* by changing their environment. This change will in turn be noted by other agents, who can then react in an appropriate way. In this paper, we focus on one such principle: synthetic pheromones, the

"virtual" variant of substances ants (among others) use to achieve a coordinated behavior on the colony level. Before explaining this in more detail, we describe the objectspaces approach in the next section, since this is the basis upon which we build.

2.1 Objectspaces

Objectspaces are a variant of an approach for concurrent computing first introduced by Linda [4], and later on refined and varied upon by several implementors (Objective Linda [13], CO^{3}PS[7]...). It remains an active research area, with notable recent variants being MARS [3] and Lime [17]. We will discuss the ideas underlying objectspaces in a fairly condensed manner here. The interested reader is referred to the aforementioned references.

The idea of objectspaces is the separation of computation and coordination. Computation is the actual functional computation the concurrent application will have to achieve, while coordination is the management of the inter-process communication, in fact transfer of the intermediary results between the concurrent processes. This concurs with the definition of coordination given earlier.

This is achieved by connecting all agents to a number of so-called *objectspaces*. The approach distinguishes between *agents*, process who are running in their own thread of control, and considered a black box by the coordination mechanism, and *passive objects*, in fact regular (e.g. Java) objects that are used as communication. Communication can only happen by putting and taking objects from the objectspace (in other words, no direct communication is possible). An objectspace is then a container for passive objects, that agents can connect to. Agents can manipulate objects in the space by executing a few primitive operations: take, put and read.

One important addition to the model is the distributed variant of the objectspace, where more than one objectspace can exist in a system. In order to find these objectspaces, agents can "travel" over them using "links". These links are nothing more than a special kind of passive objects, usually called *OS_Logicals*. An OS_Logical is a passive object that can be put in or taken out of the objectspace as usual, but in fact encapsulates a "reference" to another objectspace. Agents can connect to remote objectspaces by retrieving an OS_Logical. Agents can thus navigate between a graph of connected spaces using these OS_Logical objects.

In the next section, we elaborate on our implementation of synthetic pheromones, that is build on top of an existing objectspace architecture, CO^{3}PS[7], augmented with a meta-layer [5] that makes it easier to extend the objectspace with new mechanisms. We touch on some design and implementation issues, and evaluate the new principles on a file-sharing Gnutella-like [9] P2P system.

3 Synthetic pheromones

This section introduces the concept of synthetic pheromones, starting from principles of real pheromones and how some social insects apply them to achieve

decentralized and robust coordination. We then introduce the concept of synthetic pheromones (this name is borrowed from Parunak and Brueckner [2]).

3.1 Real Pheromones

The underlying principle of synthetic pheromones is based on the mechanism ants (among others) use to find food sources. A pheromone is a chemical substance an ant can drop in the environment. The pheromone then propagates (by Brownian motion) through the environment, as well as evaporates over time. Ants for example use it to build a path from the nest to a food source, by applying a simple algorithm:

1. if the ant smells no pheromone, wander randomly seeking a food source
2. if it does smell a pheromone, follow the trail of that pheromone
3. when food is found, pick it up and reinforce (or start) the pheromone trail to the nest by dropping additional pheromones

Two important dynamics are at work here. The first has to do with the propagation of pheromones: the stronger the trail, the further it will propagate, and the more it will attract other ants (from distances further away). While there is food remaining, ants will reinforce the trail and make it stronger, attracting even more ants, etc. This is a positive feedback loop. The second important dynamic is the evaporation of the trail: once the food is gone, no more ants will reinforce the trail, and all pheromones will evaporate, thus stop "misleading" ants. Evaporation is a "forgetting" factor for the system, and is in fact the negative feedback counterpart of the propagation mechanic.

The relatively simple mechanism of pheromones demonstrates a simple, yet effective distributed decision making mechanism. In fact, building on the example above, when a trail exists from the nest from the food source, this represents the colony's "decision" that that particular food source is worthwhile too exploit. We argue that this is an advanced form of multi-agent coordination, and aim to provide primitives in software to apply it to applications.

3.2 Synthetic pheromones

Due to the differences between the real world and the software world, the concept of pheromones needs a bit of translating to be applicable. As main idea, a pheromone in our system is represented as a passive object in an objectspace. Pheromones can thus be read from ("smelled") and put into the objectspace ("reinforced").

The data structure to represent a synthetic pheromone is fairly straightforward. Most of the ideas are from Brueckner [2]. A pheromone holds the following data:

strength this is a floating point value that is a representation of the current strength of the pheromone.

threshold this is another floating point value that indicates when the pheromone should be removed from the system. Whenever the pheromones strength falls below this threshold, it is considered completely evaporated, and will be removed. Of course, in the real world, pheromones evaporate continuously and are not suddenly removed. However, it is impossible to always hold all pheromones that have ever been put into a space, for efficiency reasons.

direction the direction the pheromone was propagated from - i.e. the direction to follow if the agent wants to follow the trail. In the real world this is not necessary since this direction is given by the gradient of the trail (closer to the nest the pheromone concentration will likely be higher since it has been more recently dropped). Adding a direction, effectively represented by an OS_Logical object representing the objectspace the pheromone was propagated from, is the most convenient solution.

properties a property map, mapping string values (the property's name) to a passive object. Usually, there will various different "flavors" of pheromones in a system. For example, ants have (among many others) a food pheromone, as well as an alarm pheromone. The pheromones properties can be used to distinguish between these various flavors. A pheromone can have as many distinct properties as is desired.

3.3 Dynamic objects

While describing the data structure of a pheromone is fairly straightforward, adding support to the system so that the pheromones dynamics (an essential part of the coordination mechanism) are also simulated is another cup of tea. We now discuss the concepts necessary to deal with these difficulties in a generic way.

The first thing to notice is that objectspaces only offer support for two kinds of objects. The first are *passive* objects, basically stale data contained in an objectspace that can only be modified by outside intervention (i.e. an agent). The second is support for *active* "objects", the agents themselves, which are processes that can act upon objectspaces but are not part of them (i.e. no agent is visible for any other agent that happens to be connected to the same space). The concept of pheromones falls somewhere in between these two concepts: a pheromone is *dynamic*, evaporating, propagating, changing - and thus agent-like; but is also a piece of *data* and must be visible in the objectspace - and thus passive object-like. Neither of these two existing concepts map on the concept of a pheromone. That is why we introduce the concept of a *dynamic object*.

A dynamic object is a special class of a passive object. Like a passive object, it can be read or taken from, and put into an objectspace. The crucial difference lies in the fact that a dynamic object can execute code before and after it is put in or taken from the objectspace.

Let's look at this in more detail. `DynamicObject` is an abstract base class that defines two methods: `preAction(ObjectSpace os,String operationName)` and `postAction(ObjectSpace os, String operationName)`. We can distinguish between two cases:

Putting the dynamic object in an objectspace: When this happens, the objectspace will first call the preAction method with the target objectspace as an argument. This method will completely execute before the dynamic object enters the objectspace. Once this is done, and the object is in, the postAction method will be called. There are no time guarantees *when* this will happen - the only guarantee is that it will be called after the object is put in. Note that changing the state (i.e. any of the instance variables) of the object in the postAction method is next to useless since these changes will never be seen by agents: the object that is already put in the space cannot be changed anymore, unless it is explicitly taken out.

Taking or reading the dynamic object from the space: When a dynamic object is taken out, *before* the object is returned to the agent, the `preAction` method will be called. Afterwards, thus after the agent is in possession of the object, the postAction method will be called. The same remark concerning changing of the object's state is valid as in the previous case.

Dynamic objects can distinguish between being put in or taken out by looking at the `operationName` parameter, that can be either "In", "Rd" or "Out".

A pheromone is then a special subclass of such a dynamic object. Propagation of the pheromone can be done after it is inserted into an objectspace. For a propagating pheromone, the `postAction` method will read all OS_Logicals from the space it is put in, and put a new version (equal to the original pheromone, with a strength decreased by a propagation factor) of itself in all these connected objectspaces. Thus a local propagation of the pheromone, from one objectspace to all objectspaces it is connected to, is achieved. Note that this propagation may trigger a chain effect: when the pheromone is thus propagated, every time it is put into a new space, a new propagation is triggered. It is crucial that the strength of the pheromone decreases with each propagation step, and that it stops propagating itself whenever it's strength falls below the threshold. This is also checked in the postAction method, and guarantees that, when the propagation factor p and threshold t are appropriately chosen (i.e. $p < 1$ and $t > 0$), the pheromone will not propagate eternally.

Evaporation is a bit trickier. Evaporation is implemented using a preAction: before it is put into an objectspace, the pheromone obtains a timestamp with the current time. Now, every time it is taken out of the space, it will reduce its own strength with a factor proportional to the difference between the time it was put in (the timestamp obtained earlier), and the current time. Once it has reached a value below it's threshold, it will remove itself from the objectspace and will not be returned to the agent - as if there is no pheromone. Once again, if the evaporation factor and the threshold are well chosen, this ensures that a pheromone will eventually disappear from the system - a sort of leasing system.

Notice that dynamic objects are more generic than pheromones - they can for example be used to achieve a multicast (similar to the propagation described above, but without changing the contents of the message). Evaluating some of their other possible uses for coordination purposes will be the subject of future work.

4 Case study: a P2P system

To validate the synthetic pheromones approach, we chose to apply it to the problem of searching data in a distributed P2P system. To this end, we built a simple simulation of such a network. Our approach is as follows. Each peer in the system is represented by a single objectspace. This objectspace contains a number of OS_Logicals, that point to the logical neighbors of that peer (the known peers). Each objectspace can also contain a number of resources, that can be of various kinds (represented by a `Resource` object in this simple example). On each objectspace peer, a query generator agent is connected that periodically generates a query for a randomly chosen resource. This generator agent simulates a user of the P2P system. Notice that, when this generator agent would be replaced by a real user, nothing else in the system would have to be changed in order to actually run the system as a peer to peer network.

The query is in fact a small mobile agent, that will wander from objectspace to objectspace, searching for its "food": a `Resource` object to match it's query. Once it has found such a resource, it will return to the peer-objectspace it came from, leaving a pheromone on every objectspace along the path it walked. Query agents that are looking around for a certain resource can check at each peer-objectspace whether there are any pheromones that may be useful for the query the agent is trying to resolve. If so, it is inclined to follow these.

The idea is then that, when some queries are done frequently, a path will emerge to a resource matching these queries, thus making the search faster, while still being able to adapt to changing circumstances. This approach is thus much similar to the "real ant" approach, and the algorithm followed by a query agent is thus equivalent to that of a real ant described in section 3.1:

1. at each node, if a matching resource is not found, read all pheromones from the objectspace that lead a resource that fulfills the agent's query (more on this later). The strength of the pheromone indicates the agent's preference to go in that direction: all strengths are summed, and the probability an agent will take the direction is equal to the strength for the direction divided by the total pheromone strength. Notice that when no pheromones are available, the agent will simply pick a direction at random.
2. if the resource is here, report that the resource is found to the originating peer, and walk back along the path followed, putting pheromones on every objectspace along the way.

To allow the agents to detect whether a pheromone is important for them or not, each pheromone has a property, called "Query". Associated with that property is the actual query the dropping agent was looking for (represented by a `Query` object). Other agents can filter on the available pheromones in a space and only take into account pheromones that match their own query. In our implementation, the query of the pheromone must be narrower than the agent's query - that is, it must match the same amount of resources or less. We chose this approach instead of representing the found resource directly in the pheromone, because the latter approach would limit the queries to criteria included in the

representation of the resource (e.g if only filenames are represented, only queries on the filename could use the pheromones). The general solution, copying the complete resource to every objectspace along the path, is obviously impractical. Selectively caching the matching resource on certain peers with high pheromone concentrations on the other hand, could be useful but is not further explored here.

4.1 Tests and results

The evaluation is still work in progress, so we only present preliminary results. Tests were done on a small network of 200 nodes, with one kind of resources, and new queries were generated at each peer for a random resource with a small time period. The network is a random network, where each node has a chance of N^z of being linked with another node, where N is the number of nodes, and z is a factor that determines the kind of network that will be the result. [1] shows that for $z = -1$ trees of all order appear, and some cycles. This seemed like a reasonable choice, with the caveat that we do not know of any studies determining this factor, and thus the structure, of real networks.

We did experiments in two kinds of environments: one where the resource the agents had to find was abundant, and the other where it was sparse. In the first setting, about 1 in 10 nodes hold the desired resource, and it is thus not very difficult to resolve a query. In the sparse resource setting, on average 1 in 100 nodes held the resource, thus representing a difficult-to-find resource.

First we tested a Gnutella-like agent as reference experiments. This agent is cloned to all possible outgoing directions at every objectspace until it reaches a hop count limit and then dies. This limit was set to three for these experiments. This agent returned a successful query result in 79% of the cases for the abundant resource setting. It found a result 13% of the time for the sparse resource setting.

For the ant-like agent, every query is resolved by a single agent, and the maximum number of objectspaces it can visit is set to fifteen (this number does not include the hops on the way back, dropping pheromones). The pheromones the agent dropped evaporate with an evaporation factor of 0.9, and propagate with a propagation factor of 0.9.

Abundant resources. Running the test 6 times revealed that 32% of the queries were successfully found, over about 400 generated queries. However, after 700 queries, the hit rate increases to 53%. It is obvious that the Gnutella agent has a much better hit rate than the ant-like agent. This is because the Gnutella agent can search a lot more nodes: since the network has an average degree (outgoing links per node) of 6 (this follows from the chosen z factor above, and the number of nodes), it searches $6^3 = 216$ peers per search. In this simple problem, this amounts to almost the complete network (not taking into account loops and isolated nodes). The ant-like agent searches only 15 peers. In this light, the result of the ant-like agent is not that bad. To verify our statement, we ran some tests where we did allow the ant-like agent to clone itself. We allowed a maximum of two clones per new objectspace (thus less than *all* the outgoing directions the gnutella agent would clone to), and reduced the maximum number

of hops from 15 to 5. As expected, this resulted in a significantly increased success percentage of 41%, after only 150 generated queries.

Sparse resources. In the sparse setting, the hit percentages were a lot lower. Only 4% of the queries was returned successfully. Allowing the ant agent to clone itself at most twice, as above, resulted in a 10% success result, which is no significant difference from the success rate of the gnutella agent.

Although in most naive cases the success percentage of the ant-like agent is lower, it can be seen that the difference can be made smaller or even non-existent if we allow the pheromones to stabilize (thus running the simulation longer), or when we allow some limited cloning of the agent. The advantage of this approach is then that it allows a better usage of bandwidth, at the cost of more storage space and CPU power on the individual peers. We think this is a good trade-off, given the fact that bandwidth is still much more expensive than storage space or CPU cycles.

To estimate the effect of the pheromones on the search, we calculated a *guidance* parameter at each peer-objectspace. This parameter is a value between zero and one. A value of zero indicates a perfect guidance, where all pheromones are zero except one (and the agent thus knows perfectly where to go). Conversely, guidance equals 1 when all pheromone strengths for all directions are equal, and agents thus choose randomly. The guidance formula is based on the formula for information entropy given by Shannon: $e = -\sum p_i log p_i$, where p_i in our application would be the probability of choosing direction i. This function is normalized to obtain values between one and zero: $g(n) = \frac{e}{log\frac{1}{n}}$, where n is the total number of possible directions. These values were seen to decrease over time as expected, and settle on a value very dependent on the particular query and network topology, as well as on parameters such as evaporation rate. The fact that the values are decreasing indicates that the longer the network is running, the better the agents get at finding queries. The speed at which this occurs is very dependent on the initial pheromone strength, and this is thus a very important parameter. The guidance was used to tune the values of the parameters, such as evaporation rate and initial pheromone strength.

The experiments conducted so far are insufficient. We feel however that the results are promising, and indicate some short term further directions in the following paragraph.

4.2 Future experiments

First of all, we will make the structure of the graph of the network more like a real network. It is known that real networks have properties that make them unlike random graphs [12] [1], such as small world properties and power-law distributions. This will make the simulation more realistic.

Further tests will be performed to assess the performance of the algorithm, as well as refinements of the algorithm. A main problem we will tackle is the controlled evaporation and propagation of pheromones. Tuning of the strength, propagation and evaporation rate is a manual process, and depends much on the

concrete problem. It is possible to automate this process somewhat by modifying these parameters on the fly using the guidance statistic. For example, if query agents consistently report low guidance, newer query agents could be sent out that drop pheromones with a higher strength or a lower evaporation rate.

Choosing a good evaporation factor is very difficult since it is very dependent on the work load of the peers: if lots of agents gather on one peer, they cannot all be scheduled at the same time. As a result, some agents may miss a pheromone because it is already evaporated before they are scheduled to execute. The proposed extension will make the system more adaptive to these kinds of problems.

5 Related Work

There is some closely related work available, concerning pheromone infrastructure and ant-like agents in P2P systems.

Brueckner describes a pheromone infrastructure upon which ours is based in [2]. This infrastructure is often used throughout the works of Parunak, Brueckner et al., e.g. [18] Their work focuses on the underlying dynamics, rather than the distributed software engineering issues focussed on in this paper. The pheromone infrastructure they describe seems mainly used in a centralized, sequential environment, and in fairly simple examples (with the notable exception of manufacturing control). We feel this work is very important to understand the underlying principles and to gain new ideas for applications. Our work is complimentary because we explicitly focus on software engineering issues in concurrent, distributed environments, which we believe to be the ultimate application environment for these agent-techniques.

AntHill [16] is a Java based framework for the design and simulation of P2P frameworks, and mainly aims to be a test bed for new P2P algorithms and approaches. The focus is also on agent-based systems, and "ant-like" agents, as the name suggests. AntHill provides the basic tools to build a P2P system, as well as a simulation tool and a genetic algorithm based optimization tool. The main difference with the work presented here is a difference in scope: AntHill aims to be a test bed and platform for P2P applications. Our goal is to provide support for distributed agent-oriented development, using P2P as an interesting case study.

Co-Fields [14] and Tuples on the Air (TOTA) [15] are two closely related approaches. Co-Fields was introduced by Tokoro [19] as a means to solve the problem of load balancing by attaching a virtual gradient field to each process. Processes can then attract or repulse each other to achieve a good spread over different hosts. Co-Fields is a variant and refinement of this approach, applying it to supply "context" to mobile agents. TOTA implements the approach for wireless ad-hoc networking applications. The main idea is that the agent's environment presents contextual information in the form of gradient fields, and that agents need only follow the fields to achieve a certain goal. Pheromones as described here can be seen as a sort of gradient field. However, the evapora-

tion of fields does not seem to be modeled in Co-Fields, and it is unclear how fields disappear from the system. Co-fields can be readily implemented using our concept of dynamic objects, which is a more general concept.

6 Conclusion

This paper is a work in progress report on the design of a coordination framework for use in large, open distributed environments. We introduced the notion of synthetic pheromones as an interesting coordination strategy, and proposed the concept of dynamic objects as a useful abstraction of these pheromones. As an evaluation, we presented an algorithm for distributed search in a P2P network, and reported on some interesting results of the simulations.

We believe the results we have presented warrant further investigation, and we will continue this work in the next few weeks. In the longer term, we will investigate other forms of coordination, and the extensions to the coordination infrastructure that are needed to allow applications to use these in a clean and modular way.

References

1. R. Albert and A.-L. Barabasi. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74(47), 2002.
2. S. Brueckner. *Return from the Ant - Synthetic Ecosystems for Manufacturing Control.* PhD thesis, Humboldt University Berlin, 2000.
3. Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, August 2000.
4. Nicolas Carriero, David Gelernter, and Jerry Leichter. Distributed data structures in linda. In *Proc. 13th ACM Symposium on Principles of Programming Languages*, 1986.
5. T. Coninx, T. Holvoet, and Y. Berbers. Using reprogrammable coordination media as mobile agent execution environments. In *ECOOP - European Conference for Object Oriented Programming - 8th Workshop on Mobile Object Systems*, 2002.
6. Di Caro G. and M. Dorigo. AntNet: A mobile agents approach to adaptive routing. Technical Report IRIDIA/97-12, Universit Libre de Bruxelles, Belgium, 1997.
7. Tom Holvoet. *An Approach for Open Concurrent Software Development.* PhD thesis, Department of Computer Science, KULeuven, Belgium, 1997.
8. The Autonomic Computing Homepage. http://www.research.ibm.com/autonomic, 2003.
9. The Gnutella homepage. http://gnutella.wego.com/, 2002.
10. Oliver Huber. Mobile agents in active networks. In *ECOOP'97 Workshop Mobile Object Systems*, Finland, June 1997.
11. N. R. Jennings, J. M. Corera, and I Laresgoiti. Developing industrial multi-agent systems. In *Proceedings of the First International Conference on Multi-agent Systems, (ICMAS-95)*, pages 423–430, 1995.
12. M. Jovanovic, F.S. Annexstein, and K.A. Berman. Modeling peer-to-peer network topologies through "small-world" models and power laws. In *TELFOR*, Belgrade, Yugoslavia, November 2001.

13. Thilo Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming.* PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997.
14. M. Mamei, F. Zambonelli, and L. Leonardi. A physically grounded approach to coordinate movements in a team. In *1st International Workshop on Mobile Teamwork (at ICDCS).* IEEE CS Press, 2002.
15. Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Tuples on the air: a middleware for context-aware computing in dynamic networks. Technical Report DISMI-2002-24, University of Modena and Reggio Emilia, august 2002.
16. Alberto Montresor. Anthill: a framework for the design and analysis of peer-to-peer systems. In *4th European Research Seminar on Advances in Distributed Systems*, Bertinoro, Italy, May 2001.
17. Amy Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: a middleware for physical and logical mobility. In *Proc. of the 21th International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001.
18. John A. Sauter, Robert Matthews, H. Van Dyke Parunak, and Sven Brueckner. Evolving adaptive pheromone path planning mechanisms. In Christiano Castel-Franchi and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems.* ACM Press, 2002.
19. M. Tokoro and K. Honda. The computational field model for open distributed environments. *Concurrency: Theory, Language, and Architecture*, LNCS(491), 1991.