A Pheromone-Based Coordination Mechanism Applied in Peer-to-Peer

Kurt Schelfthout and Tom Holvoet

Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A,
3001 Leuven, Belgium
{Kurt.Schelfthout, Tom.Holvoet}@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 scalable, distributed systems, such as peer to peer systems. We present a software 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 distributed systems 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 [1](networks that are able to dynamically reconfigure themselves in response to faults or changing quality of service requirements) and manufacturing control [2](where it is difficult to gather and plan centrally because information may be outdated very rapidly).

These systems seem like a natural application domain for multi-agent systems (hereafter written MAS). An agent is 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.

Our interest is in 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 typical example is the division of work among autonomous agents, and the reconstruction of the end result after the work is done. Coordination is especially important in the kind of systems mentioned above, since they are inherently composed out of multiple distributed entities that need to cooperate for the system to function.

Our 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 can be noted by other agents, who are then able to react appropriately. This is interesting because agents thus become highly *decoupled*: in order to communicate, the sending agent does even not need to be alive when the receiving agent gets the message.

An existing approach for coordinating agents is the tuplespace approach, first introduced for concurrent computing in Linda [3], and later on refined and varied upon, notably for object-oriented and distributed computing (Objective Linda [4], CO³PS[5] . . .). It remains an active research area, with notable recent variants being MARS [6] and Lime [7].

Presented here is an extension of objectspaces with support for synthetic pheromones, the virtual variant of substances ants (among others) use to achieve a coordinated behavior on the colony level. This is discussed in detail in Section 2. In the following section, we elaborate on our implementation of synthetic pheromones, that is build on top of an existing objectspace architecture, CO³PS[5], augmented with a meta-layer [8] that makes it easier to extend the objectspace with new mechanisms. We indicate some design and implementation issues. Furthermore, the new principles are evaluated on a file-sharing Gnutella-like [9] P2P system.

2 Pheromones as Dynamic Objects

The underlying principle of synthetic pheromones [10] is based on the mechanism ants (among others) use to find food. 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 laying pheromone trails to the food.

These trails are governed by two important dynamics. The first is the propagation of pheromones: the stronger the trail, the further it propagates, and the more it attracts other ants (from distances further away). While there is food remaining, ants keep on reinforcing the trail, attracting even more ants. The second important dynamic is the evaporation of the trail: once the food is gone, no more ants are reinforcing the trail, and all pheromones eventually evaporate, thus stop “misleading” ants. Evaporation is a “forgetting” factor for the system. This keeps the system responsive to changes. This scheme appears to be very robust and scalable in practice. Therefore, we would like to apply it as a coordination mechanism in software, as for example in [10].

A pheromone in our (software) system is represented as a passive object in an objectspace. Pheromones can thus be read from and put into the objectspace. Also, our ants (agents) live in a graph environment, each node being an objectspace, connected to its neighbors using link objects (a special kind of passive objects).

The data structure to represent a synthetic pheromone is fairly straightforward. The two most important values are a current *strength* of the pheromone

(used to discriminate between two pheromone paths - the stronger is the most likely to be followed) and a *direction*, which indicates the direction of the trail. Another property is a *threshold* value. When the strength falls beneath this threshold, the pheromone is deleted from the system.

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. Traditional objectspace technology only provides passive objects, that can be put into the objectspace, and active objects, the agents themselves. There is no support for what we call *dynamic objects*: objects that can be put into an objectspace but that can change over time without agent intervention. This is achieved by allowing dynamic objects to execute some code before and after they are put into an objectspace. So, they can change both themselves and any objectspace before and after they are put in or taken out of an objectspace.

An implementation was done in Java as a meta-layer upon CO³PS[5] as follows. `DynamicObject` is an abstract base class that defines two methods: `preAction(ObjectSpace os, String operationName)` and `postAction(ObjectSpace os, String operationName)`. When a dynamic object is *put in* an objectspace, the objectspace first calls the `preAction` method with the target objectspace as an argument. This method executes completely before the dynamic object enters the objectspace. Once this is done, and the object is in, the `postAction` method is called. When a dynamic object is *taken or read* from the space, *before* the object is returned to the agent, the `preAction` method is called. Afterwards, thus after the agent is in possession of the object, the `postAction` method is called. Dynamic objects can distinguish between being put in or taken out by looking at the `operationName` parameter, that can be either "Take", "Read" or "Put".

A pheromone is then a 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 gets all the objectspaces the target objectspace is connected to, and puts 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 is achieved. Since this can trigger a chain effect, it is crucial that the propagation factor is smaller than one. This ensures that the pheromone's strength will eventually fall below its threshold, thus removing it from the system.

Evaporation is implemented using a `preAction`: before it is put into an objectspace, the pheromone obtains a time stamp with the current time. Now, every time it is taken out of the space, it reduces 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 its threshold, it removes itself from the objectspace and thus is not returned to the agent - as if there is no pheromone. Once again, if the evaporation factor is well chosen a pheromone will eventually disappear from the 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.

3 Case Study: A P2P System

To validate the synthetic pheromones approach, we applied it to the problem of searching data in a distributed P2P system. To this end, we built a simple simulation of such a network. Each peer in the simulation is represented by a single objectspace. This objectspace contains a number of passive objects, 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.

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

The idea is then that, when some queries are performed 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. The algorithm followed by a query agent is the following:

1. at each node, if a matching resource is not found, read all pheromones from the objectspace that lead to a resource that fulfills the agent's query. To this end, each pheromone has a **Query** object associated with it, representing the query the dropping agent was looking for. Agents can filter on the available pheromones in a space and only take into account pheromones that match their own query. The strength of the pheromone indicates the agent's preference to go in that direction: all strengths are summed, and the probability an agent takes the direction is equal to the strength for the direction divided by the total pheromone strength. When no pheromones are available, the agent simply picks a direction at random.
2. if the resource is here, report to the originating peer that the resource is found, and walk back along the path followed, putting pheromones on every objectspace along the way.

Tests and results. Tests were done on a small network of 200 nodes, with one kind of resources, and new queries for a random resource were generated periodically at each peer.

We did experiments in two kinds of environments: one where the resource the agents had to find was abundant (resource was available in 1 in 10 nodes), and the other where it was sparse (resource in 1 in 100 nodes). First we used a Gnutella-like search 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. 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.

In the abundant resources setting 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: by construction, the network has an average degree (outgoing links per node) of 6, so 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 fairly good. To verify this 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 an increase from 32% to 41% successes.

In the sparse resources 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 10% successes, which is no significant difference from the success rate of the Gnutella agent.

Although in most 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.

Discussion. 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.

Choosing a good evaporation factor is difficult since it is 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.

Another problem to face is the problem of scalability. Dropping pheromones at each peer may result in a blowup of the space required at each space. There is a trade off here: one can limit this space by setting a larger evaporation factor (meaning pheromones will evaporate quicker), however it will become more difficult for the agents then to put down a lasting path (i.e. the search will have to be much more popular). Another way to limit space complexity is to collapse specific searches into more general searches (e.g. `foo*.jpg` into `f*.jpg`). Again, some information is lost, yet storing too much information may lead to an unmanageable system.

4 Conclusion

This is a 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 results of the simulations.

References

1. Di Caro, G., Dorigo, M.: AntNet: A mobile agents approach to adaptive routing. Technical Report IRIDIA/97-12, Université Libre de Bruxelles, Belgium (1997)
2. Jennings, N.R., Corera, J.M., Laresgoiti, I.: Developing industrial multi-agent systems. In: Proceedings of the First International Conference on Multi-agent Systems, (ICMAS-95). (1995) 423–430
3. Carriero, N., Gelernter, D., Leichter, J.: Distributed data structures in linda. In: Proc. 13th ACM Symposium on Principles of Programming Languages. (1986)
4. Kielmann, T.: Objective Linda: A Coordination Model for Object-Oriented Parallel Programming. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany (1997)
5. Holvoet, T.: An Approach for Open Concurrent Software Development. PhD thesis, Department of Computer Science, KULeuven, Belgium (1997)
6. Cabri, G., Leonardi, L., Zambonelli, F.: Mars: A programmable coordination architecture for mobile agents. *IEEE Internet Computing* **4** (2000) 26–35
7. Murphy, A., Picco, G.P., Roman, G.C.: Lime: a middleware for physical and logical mobility. In: Proc. of the 21th International Conference on Distributed Computing Systems (ICDCS-21). (2001)
8. Coninx, T., Holvoet, T., Berbers, Y.: Using reprogrammable coordination media as mobile agent execution environments. In: ECOOP - European Conference for Object Oriented Programming - 8th Workshop on Mobile Object Systems. (2002)
9. The Gnutella homepage: <http://gnutella.wego.com/> (2002)
10. Brueckner, S.: Return from the Ant - Synthetic Ecosystems for Manufacturing Control. PhD thesis, Humboldt University Berlin (2000)