A Scalable Hash-Based Mobile Agent Location Mechanism

Georgia Kastidou, Evaggelia Pitoura
Department of Computer Science
University of Ioannina, Greece
{georgia, pitoura}@cs.uoi.gr

George Samaras
Department of Computer Science
University of Cyprus, Cyprus
cssamara@ucy.ac.cy

Abstract

In this paper, we propose a novel mobile agent tracking mechanism based on hashing. To allow our system to adapt to variable workloads, dynamic rehashing is supported. The proposed mechanism scales well with both the number of agents and the number of moving and querying operations. We also report on its implementation in the Aglets platform and present performance results.

1. Introduction

Mobile agents are processes that may be dispatched from a source computer and be transported to remote servers for execution. The driving force motivating mobile agent-based computation is twofold. First, mobile agents provide an efficient, asynchronous method for searching for information or services in rapidly evolving networks: mobile agents may be launched into the unstructured network and roam around to gather information. Second, mobile agents support intermittent connectivity, slow networks, and light-weight devices.

In any mobile agent system, the ability to communicate with agents in real-time, as agents move from one network node to another, is essential for retrieving any data or information that they have collected, and for supporting coordination and cooperation among them. Communication with a mobile agent subsumes the ability to locate it (i.e., find the node and execution environment in which it currently resides). Locating agents efficiently is thus an issue central to any mobile agent system.

Mobile agents systems are highly-dynamic open systems in which the number of agents varies considerably over time as new agents are created and existing agents die. A location schema in such systems should scale well with the number of agents and their distribution and mobility. In this paper, we present such an agent location mechanism. Special agents, called Information Agents (IAgents), maintain the current location of a set of mobile agents assigned to them. The assignment of mobile agents to IAgents is based on a system-wide hash function and thus is very efficient. The IAgents are also mobile agents whose location depends on the distribution of the mobile agents that they serve. By separating between these two issues (number and location of the IAgents), we are able to treat each one of them differently and apply mechanisms appropriate for each one of them.

To allow our system to adapt to the changing number of mobile agents and the variable system workload (i.e., the mobility rate of the agents and the rate of requests for communication), the number of IAgents changes over time. In particular, when an IAgent is over-loaded, it splits its load by creating a new IAgent. Analogously, under-loaded IAgents are merged by assigning their load to other existing IAgents. The process of splitting and merging IAgents should not affect the mapping of mobile agents to IAgents that are not involved in the process. To this end, our hash function is extendible. Our mechanism is independent of any specific agent-based platform since the mapping of mobile agents to IAgents is not based on any particular agent-naming scheme.

We have expanded Aglets [7], a mobile agent infrastructure, with our location mechanism. Our performance results show that our mechanism scales well when compared with a centralized location schema. In particular, when configured appropriately, it takes almost constant time to locate an agent independently of the system workload.

The remainder of this paper is structured as follows. In Section 2, we provide an overview of our approach. In Section 3, we introduce our hash function, while in Section 4, we describe rehashing: the procedure of dynamically adjusting the hash function for load balancing. In Section 5, we present an implementation of our mechanism in Aglets and its performance. Finally, we present related work in Section 6 and our conclusions in Section 7.

2. Our Hash-Based Approach

2.1 Overview of our Location Mechanism

We propose a two-tier mechanism. Special agents, called Information Agents (IAgents), are responsible for maintaining the current location of a set of mobile agents. Which set of mobile agents is associated with each IAgent is determined through a hash function. This mapping changes over time as new IAgents are created or existing IAgents are merged depending on the current system workload.

To locate a mobile agent A, in the first phase, we determine which special agent (IAgent) is responsible for maintaining the precise current location of A. This is done by applying the hash function on A's id. In the second phase, the responsible IAgent is contacted.

For our system to adapt to the changing number of mobile agents and the varying system workload, the number of IAgents changes over time. Specifically, when an IAgent becomes over-loaded, it splits its load by creating a new IAgent. Similarly, under-loaded IAgents are merged by assigning their load to other existing IAgents. Thus, the mapping of mobile agents to IAgents should be dynamically adjustable as well. However, the splitting and merging process should affect the mapping of only the mobile agents and the IAgents that are involved in the process. To this end, we choose a dynamic hash function.

Another basic characteristic of our mechanism is the maintenance of the hash function. There is a central static agent (HAgent) that keeps the current hash function. Every time the hash function changes, the copy of the HAgent is immediately updated (primary copy). For reasons of efficiency, copies of this hash function are maintained locally in every node of the system. These copies may be temporally out-of-date (secondary copies).

2.2 System Components

The basic agents that constitute our location management mechanism are: (i) IAgents (Information Agents), (ii) LHAgents (Local Hash Agents), and (iii) the HAgent (Hash Agent).

The *IAgents* are mobile agents that maintain information about the current location of the mobile agents that are assigned to them. Every IAgent maintains for each mobile agent it serves its id and its precise current location. The location of an IAgent depends on the current location of the agents it serves and may change over time.

The LHAgents and the HAgent are responsible for the maintenance of the hash function. In particular, there is one *LHAgent* at each node of the system. Each LHAgent maintain a local copy of the hash function. The *HAgent* is the agent (mobile or static) that maintains the primary copy of the hash function. The HAgent is also responsible for coordinating the splitting and merging processes.

2.3 Basic Operations

Agent Movement

During its creation, each mobile agent A communicates with the LHAgent at its node to find out the id and the current location of the IAgent that is responsible for maintaining its current location. Subsequently, each time A moves, it informs its IAgent about its new location.

Locating an Agent.

Each time, an agent Q wants to communicate with another agent A, it communicates first with its own local LHAgent and gets the id and the current location of A's IAgent. Then, Q queries the specified IAgent for the current location of A. Upon receiving the query, the IAgent checks whether it is still responsible for A (the IAgent may have seize to serve A, if the hash function has been modified). If it still servers A, it replies to Q with A's current location. Otherwise, it notifies Q that it is no longer responsible for A. This will trigger the hash function update propagation procedure described in Section 4.3.

3. Description of Hashing

For attaining scalability and adaptability to the changing system conditions, the number of IAgents changes over time. The mapping of agents to IAgents is through a hash function *H*. Since this mapping changes dynamically over time, we choose *H* from the category of extensible hash functions [6]. Function *H* takes as input the binary representation of a mobile agent's id and returns the id of the IAgent that is responsible for this agent. Specifically, *H* uses some prefix of the binary representation of the agent's id. The size (i.e, the number of bits) as well as which bits of the prefix are used varies over time.

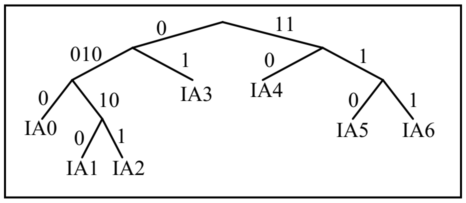

Figure 1: Hash Tree

To represent the hash function *H*, we use a binary tree that we call a *hash tree*. Figure 1 depicts an instance of such a hash tree. With each edge of the hash tree, we associate a label. A label is a string of bits. The first bit of the label of each edge *(u, v)* determines whether node *v* is on the left or the right of *u*. If *v* is on the left of *u*, the first bit of the label is "0"; otherwise it is "1". We call the first bit of each label its *valid bit*. The multi-bit labels are the result of splitting and merging IAgents.

The concatenation of the labels of all edges in the path from the root of the hash tree to a leaf node v is called the *hyper-label* of the leaf node v. For example, in Figure 1, the hyper-label of leaf "IA2" is 0010101.

For ease of presentation, we shall use the character "." to separate the labels in each hyper-label. For example, hyper-label 0010101 will be denoted 0.010.10.1.

Each leaf node of the hash tree corresponds to an IAgent. The IAgent at a leaf node v keeps information for the current location of the mobile agents for which the prefix of the binary representation of their id is compatible with the hyper-label of v. A prefix of the binary representation of a mobile agent's id is *compatible* with a hyper-label, if and only if the valid bit of each label in the hyper-label is equal to the k-th character of the binary representation, where k is the position of the specific valid bit in the hyper-label.

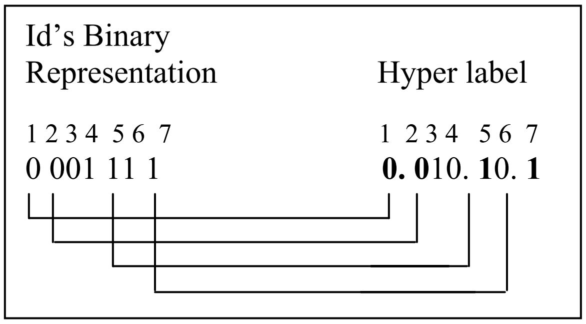

Figure 2: Compatibility between prefixes and hyper-labels (with bold letters are the valid bits)

For instance, prefix 0001111 is compatible with the hyper-label 0.010.10.1, since the valid bit of all labels in the hyper-label match with the corresponding bits of the prefix (Figure 2). This also means that agent with prefix 0001111 is mapped to IAgent "IA2" (Figure 1).

In other words, to match agents with IAgents we just use the valid bits of each label. For example, in the hash tree shown in Figure 1, the IAgent with Id "IA3" serves all agents with prefix 01, while the IAgent with Id "IA5" serves all agents with prefix 1x10, where x can be either 1 or 0.

This leads us to the following simple procedure for finding the IAgent that serves a specific agent A. First, A's id is converted into its binary representation. Then, the hash tree is traversed as follows. Starting from the root of the hash tree, we proceed towards a leaf of the hash tree by checking one by one the bits of the binary representation. If the value of the bit is 1, we go to the right child of the node; otherwise we go to node's left child. In the case where a label of an edge has *k* bits, where *k* > 1, we ignore the next *k - 1* bits of the binary representation and the next selection of a node is based on the bit following these k-bits.

4. Rehashing

By rehashing we refer to the procedure during which the hash function changes. The hash function changes when the structure of the hash tree is modified because of the deletion or insertion of an IAgent.

The main purpose of the insertion or the deletion of an IAgent is the re-organization of the hash tree structure in order to uniformly distribute the load created by the requests either for locating or updating an agent's current location. Specifically, we guarantee that the rate of requests received by each IAgent does not exceed a T_{max} or falls below a T_{min} threshold. To compute the current workload, we maintain running statistics of the requests received by each IAgent.

The process of creating a new IAgent or merging an existing one is coordinated by the HAgent. The HAgent ensures that only one such process is in progress at each time.

4.1 Creating New IAgents

When the rate of the messages that an IAgent receives exceeds the T_{max} threshold, a new IAgent is created so that the load is split. To distribute the load among IAgents fairly, each IAgent maintains statistics regarding the access load of each agent it serves.

The statistics maintained may vary in their level of detail leading to different heuristics for efficient rehashing. For example, we may maintain the exact number of update and query requests received per agent or for groups of agents (e.g., all agents with a specific prefix). In this paper, we assume that we maintain for each agent the accumulated rate of update and query requests.

The splitting procedure is based on the fact that only the valid (i.e. first) bit of a label is used when determining the mapping between agents and IAgents. Thus, when a label has more than one bit, we could use the other unused bits of the label to extend hashing. Doing so would result in more balanced hash trees or in other words in using shorter prefixes. This observation leads to two different forms of splitting. In the first case (simple split), all labels in the hyper-label of the IAgent have one bit. In the other case (complex split), there is at least one label at the hyper-label having more than one bit.

Simple split is performed when all labels in the hyper-label of the IAgent A to be split have exactly one bit. In this case, we split the load by using m ($m \geq 1$) extra bits of the prefix. Specifically, IAgent A starts by using m = 1 bit to split its load. If this results to an uneven split, A increments m by one, and tries to split its load using m = 2 bits. This procedure continues until m is sufficiently large to produce an even split of A's load. In terms of the hash tree, we create two new leaf nodes as children of the node to be split. The last label of the hyper-label of A is augmented with m – 1 bits, reflecting the fact that the split was done on the m-th bit.

For instance, assume that we need to split the IAgent "IA3" in Figure 1. Its hyper-label is 0.1. Let m = 1. We create two new nodes in the hash tree, one with hyper-label

0.1.0 that corresponds to the existing IAgent “IA3” and one with hyper-label 0.1.1 for the new IAgent (Figure 3).

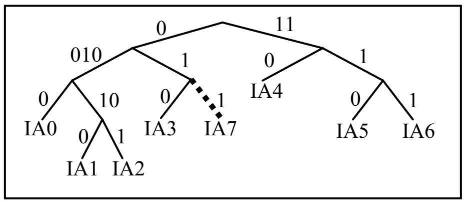

Figure 3: **Simple Split**

Complex split occurs when there is at least one label in the hyper-label of the node to be split that has more than one bit. In this case, we use these bits for splitting. The motivation is to use the unused bits in the label to produce more balanced hash trees. If this fails to distribute the workload among the IAgents, then we switch to simple split that always splits a leaf node. In particular, we start by considering the left-most multi-bit label of the hyper-label. We start by considering the first bit after the valid bit. If this results in an uneven split, we use the second bit and so on. If the attempt to split based on the leftmost multi-bit label fails, we consider the next multi-bit label. This procedure continues until a successful split is possible. If this is not possible, we switch to simple split.

For example, say we want to split IAgent “IA1” with hyper-label 0.010.10.0. We start the splitting based on the first bit of label 010. Assume this is successful. Then, we create two new nodes, one with hyper-label 0.0.10.10.0 and one with hyper-label 0.0.0 (Figure 4).

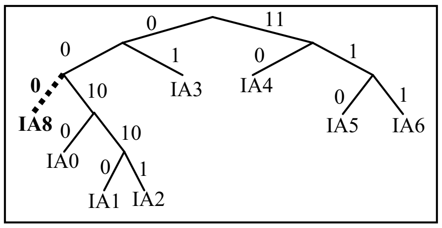

Figure 4: **Complex Split**

4.2 Merging IAgents

If the rate of messages that an IAgent receives falls below the T_{min} threshold, we merge this IAgent with existing IAgents. The agents served by the merged IAgent are assigned to some other IAgents of the system.

Similarly with the case of split, we consider two different cases. In the first case, *simple merge*, the sibling in the hash tree of the IAgent is a leaf. In this case, we simply merge the node with its sibling. For instance, assume that IAgent “IA6” in Figure 1 needs to be merged. It is merged with its sibling “IA5” (Figure 5).

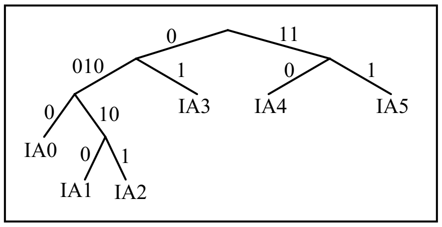

Figure 5: **Simple Merge**

In *complex merge*, the sibling of the IAgent to be merged is an internal node. In this case, the load of the IAgent is assigned to the IAgents at the subtree rooted at its sibling node. For example, assume that IAgent “IA0” in Figure 1 is merged. This results in the hash tree in Figure 6.

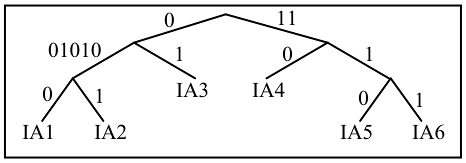

Figure 6: **Complex Merge**

Merging may lead to reducing the height of the hash tree. It may also result in overloading some of the IAgents that are assigned the agents that were previously served by the IAgent that was merged. In this case, the overloaded IAgents may need to be split.

4.3 Hash Function Update Propagation

When the hash function is modified, only the HAgent is updated immediately. The local copies of the hash function at the LHAgents are updated on demand. An update of the hash function is initiated either by (i) an agent that has moved and contacts the wrong IAgent for updating its current location or (ii) an agent that is searching for another agent and contacts the wrong IAgent. In both these cases, the mobile agent or the querying agent respectively contacts their local LHAgent which in turn contacts the HAgent to get the updated copy of the hash function.

5. Implementation and Performance Results

For the evaluation of the efficiency of our mechanism, we implemented it in the Aglets 2.0.1 mobile agent platform [7]. To study its scalability, we also implemented in Aglets a centralized mechanism. In the centralized scheme, there is a single central agent that is responsible for maintaining the current location of all mobile agents in the system. This central agent performs the same functions as the IAgents in our system.

To evaluate our mechanism, we present two experiments that compare the scalability of our mechanism with the scalability of the centralized scheme. The first experiment evaluates the performance in relation to the number of mobile agents, while the second with the mobility rate of the mobile agents. In both cases, we consider as our performance metric the average response time of a query for the location of a mobile agent (TAgent) selected randomly from all the mobile agents in the system. We call this location time. The total number of queries is 200 in each case.

The T_{max} and T_{min} values were set at 15 and 5 messages per second respectively. These values depend on various parameters, such as the type of nodes that host the IAgents as well as the agent implementation platform. We found out that these values work well in our setting. Developing heuristics for setting these values is part of our plans for future work.

The experiments were performed in real time conditions on a LAN network using 10 Sun Blade 100 running Solaris 2.8. Each experiment was run multiple times and we report the statistically normalized averages.

For the first experiment, we consider constant the mobility rate of the TAgents and change their number. The number of the TAgents that we consider is 10, 20, 30, 50 and 100. Each TAgent stays at each node for 0.5 sec. As shown in Figure 7, in the centralized scheme, the time to locate a TAgent increases linearly with the number of TAgents as opposed to our mechanism in which the location time stays almost constant.

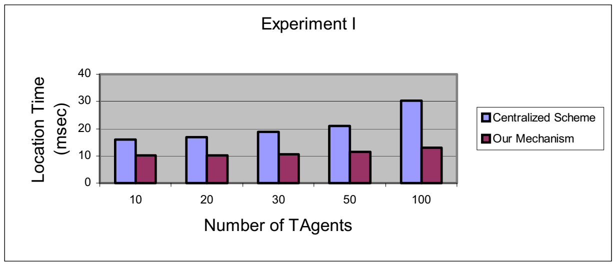

Figure 7: Results of Experiment I

For the second experiment, we consider constant the number of TAgents in the system and modify their mobility rate, i.e., how long they stay at each node. The faster the Tagents move, the more update messages the IAgents receive. In this experiment, we consider a small number of TAgents (20) to emphasize the effect of mobility. We consider that each TAgents remains at each node for 100, 200, 500, 1000 and 2000 msecs. As shown in Figure 8, our mechanism outperforms the centralized one.

Although, it was expected that the time for locating a mobile agent with our mechanism would be much smaller than with the centralized approach, it is interesting to note that this time remains almost constant regardless of the current system conditions. In other words, if at some point a large number of mobile agents is created in the system or their moving rate changes unpredictably, our mechanism will adapt nicely by changing appropriately the hash function and deleting or inserting new IAgents in order to keep constant the time needed to locate a mobile agent.

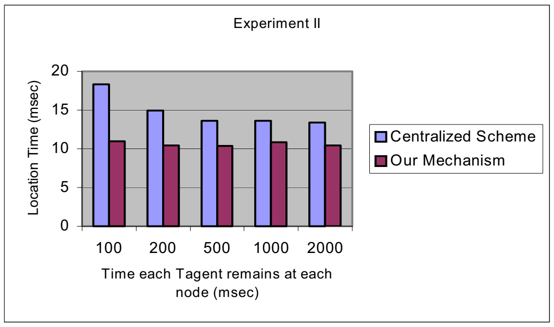

Figure 8: Results of Experiment II

6. Related Work

The problem of locating mobile objects is a well-studied one [2]. However, although, in most mobile agents platforms, knowing the precise current location of the receiver is considered necessary for inter-agent communication, many of them (e.g., Aglets [7], Mole [9], D'Agents [10], Concordia[11], and Grasshoper[12]) do not provide an agent location mechanism. Scalability, although important, is also an issue rather under-researched in the context of agent platforms [4].

Ajanta's location mechanism [5, 8] implements an HLR/VLR scheme [2] in which a registry keeps information for the agents which are currently located in its domain. In addition, each registry maintains the precise current location for the agents which were created in its domain. One limitation with Ajanta is that the name of each agent contains information about the registry in which the agent was created. Thus, this mechanism cannot be used in agent systems that use a naming system that does not contain such information.

Voyager [13] implements a centralized schema with forwarding pointers. In this scheme, every agent that wishes to be located by other agents registers to one or more name services. Each time an agent moves, it informs all the name services to which it has registered. To locate an agent, one must know either a name service to which the agent has registered or (under some circumstances) a node that the agent has visited during its trip (these nodes will forward the request until the agent is reached).

As far as we know, ours is the first approach that considers dynamic hashing in the context of mobile agents. Hashing was also proposed for locating agents in [14]. In this work, the emphasis is on security; mobile agents are assigned to tracking agents (IAgents) by means of a cryptographic hash function.

Hashing has been used to map data items to servers in many domains. A distributed variant of an extendible hashing data structure is presented in [3]. The proposed data structure consists of buckets of data that are spread across multiple servers. Chord [1] is a protocol that uses hashing for locating the node in a peer-to-peer system that stores a particular data item. The hash function used in Chord is a variant of consistent hashing. Consistent hashing distributes data items to nodes so that each node receives roughly the same number of items. However, in our case, our goal is to balance the total workload received at each node as opposed to the number of items.

One issue that was not considered in this paper is guaranteed agent discovery; that is, ensuring that the location of an agent is found even if an agent moves faster than the requests for its location. This issue is the topic of [15, 16] and is an important direction for future work.

7. Conclusions

In this paper, we propose a hash-based approach to the problem of locating mobile agents. Special agents, called IAgents, maintain the current location of a set of mobile agents. Which set of mobile agents is associated with each IAgent is determined through a hash function. This association changes over time as new IAgents are created or existing IAgents are merged depending on the system workload. Our experiments show that our approach scales well with both the number of agents and their mobility rate and provides almost constant search time for locating an agent independently of the system workload.

We are currently extending our system in two ways. First, we study a dual problem, the placement of the IAgents so that locality is exploited. For example, the IAgents could move closer to the majority of the agents that they serve. Second, we investigate means for enhancing the fault tolerance of our mechanism. Currently, we are supporting a primary copy mechanism for the hash function, thus making the HAgent that keeps this copy a vulnerability point.

Acknowledgements

This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32645.

References

[1] Castro, P., Greenstein, B., Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H., Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications, In Proc. ACM SIGCOMM 2001.

[2] Pitoura E.,and Samaras, G., Locating Objects in Mobile Computing, IEEE Transactions on Knowledge and Data Engineering. Vol. 13, No. 4, pp. 571 – 592, July/August 2001.

[3] Hilford, V., Bastani, F.B., and Cukic, B., EH* - Extendible Hashing in Distributed Environment, In Proc. of the COMPSAC '97 – 21st International Computer Software and Applications Conference.

[4] Brazier, F., van Steen M., and Wijngaards, N., On MAS Scalability. In Proc. Of the 5th International Conference on Autonomous Agents, Montreal, Canada, May 28 – June 01, 2001.

[5] Karnik, N. and M., Tripathi, A.,R., Design Issues in Mobile Agent Programming Systems, IEEE Concurrency. Vol. 6, No. 3, pp. 52-61, July – September 1998.

[6] R.J., Enbody, and Du, H.C., Dynamic Hashing Schemes., ACM Computing Surveys. Vol 20, No. 2, June 1988.

[7] Lange, D.B., and Oshima, M., Programming and Deploying Java Mobile Agents with Aglet, Addison Wesley, 1998

[8] Tripathi, A., Karnik, N., Ahmed, T., Singh, R., Prakash, A., Kakani, V., Vora, and M., Pathak, M., Design of the Ajanta System for Mobile Agent Programming, Journal of System and Software, May 2002

[9] Baumann, J., Hohl, F., Rothermel, and K., Straber, M., Mole – Concepts of a Mobile Agent System, WWW Journal, Special issue on Applications and Techniques of Web Agents, volume 1, no 3, 1998

[10] D'Agents http://agent.cs.dartmouth.edu/

[11] Mitsubishi Electric ITA, Concordia Developer's Guide, October 1998

[12] IKV++ GmbH, Release 2.2, Grasshoper Programmer's Guide, March 2001

[12] ObjectSpace Voyager: Technical overview, Dec. 1997. http://www.objectspace.com/voyager/whitepapers/VoyagerTechOview.pdf.

[14] Roth V. and Peters J., A Scalable and Secure Global Tracking Service for Mobile Agents, In Proc. Mobile Agents 2001, Vol. 2240 of LNCS. Springer Verlag, December 2001.

[15] Moreau L. Distributed Directory Service and Message Router for Mobile Agents. Science of Computer Programming, 39(2-3):249-272, 2001.

[16] Murphy A. L. and Picco G. P., Reliable Communication for Highly Mobile Agents. In Journal of Autonomous Agents and Multi-Agent Systems, Special issue on Mobile Agents. Danny Lange ed., (to appear).