A Peer-to-Peer Approach to Resource Discovery in Multi-agent Systems

Vassilios V. Dimakopoulos and Evaggelia Pitoura

Department of Computer Science, University of Ioannina
GR 45110 Ioannina, Greece
{dimako,pitoura}@cs.uoi.gr

Abstract. A multi-agent system is a network of software agents that cooperate to solve problems. In *open* multi-agent systems, the agents that need resources provided by other agents are not aware of which agents provide the particular resources. We propose a fully distributed approach to this resource discovery problem. Each agent A maintains a limited size local cache in which it keeps information about k different resources, that is, for each of the k resources, it stores the contact information of one agent that provides it. The agents in the cache of agent A are called A's neighbors. An agent searching for a resource contacts its local cache and if there is no information for the resource, it contacts its neighbors, which in turn contact their neighbors and so on until the resource is found in some cache. We consider variations of this flooding-based search and develop and verify by simulation analytical models of their performance for both uniformly random resource requests and for requests in the case of hot spots. Finally, we introduce two approaches to the problem of updating the caches: one that uses flooding to propagate the updates and one that builds on the notion of an inverted cache.

1 Introduction

In a multi-agent system (MAS), agents cooperate with each other to fulfill a specified task. As opposed to *closed* MAS where each agent knows all other agents it needs to interact with, in *open* MAS such knowledge is not available. To locate an agent that provides a particular resource, most open MAS infrastructures follow a central directory approach. With this approach, agents register their resources to a central directory (e.g., a middle agent [14]). An agent that requests a resource contacts the directory which in turn replies with the contact information of some agent that provides the particular resource. However, in such approaches, the central directories are potential bottlenecks of the system both from a performance and from a reliability perspective.

In this paper, we advocate a new approach to the resource discovery problem in open MAS inspired by search procedures in peer-to-peer systems. Each agent maintains a limited size local cache with the contact information for k different resources (i.e., for each of the k resources, one agent that offers it). This results in a fully distributed directory scheme, where each agent stores part of the directory. The agents in the cache of an agent A are called its *neighbors*; they are the agents that A knows about. We model this system as a directed graph. Each node of the graph corresponds to an agent and there is a directed edge from a node A to all its neighbors.

We consider flooding-based approaches to searching for a resource. An agent that searches for a particular resource checks the entries of its local cache. If there is no information for the resource, the agent contacts its neighbors which in turn check their own caches and if no information is found, they contact their neighbors. This search procedure continues until either the resource is located in some agent's local cache or a maximum number of steps is reached. If the resource cannot be found, the agent has to resort to some other (costly) mechanism (e.g., to a middle agent) which is guaranteed to reply with the needed information. We provide a number of variations of the search procedure based on which subset of an agent's neighbor is contacted at each step.

Caching can be seen as complementary to directories. Small communities of agents knowing each other can be formed. Such a fully distributed approach eliminates the bottleneck of contacting a central directory. It is also more resilient to failures since the malfunction of a node does not break down the whole network. Furthermore, the system is easily scalable with the number of agents and resources.

We provide analytical estimations of the performance of the proposed search procedures and validate them by simulation. We study both uniformly random requests and requests in the case of hot spots. In the former case, we assume that the entries in the cache are random, that is the entries of each cache is a uniformly random subset of the available resources. In the latter case, we assume that some resources (i.e., the hot spots) appear in a large number of caches.

We also consider the problem of cache updates. We outline two approaches. One approach is based on the notion of an inverted cache: in addition to its local cache, each agent A maintains a list of the agents that have A as their neighbor (i.e., the agents that have A's contact information in their caches) and uses this cache to propagate the updates. The other approach is symmetric to the search procedure: when an agent either changes its location or its resources, it propagates these updates to its neighbors, which in turn contact their neighbors.

The remainder of this paper is structured as follows. In Section 2, we present local caches, in Section 3, search procedures and their analysis and in Section 4, cache updates. A summary of related work is given in Section 5, while conclusions are provided in Section 6.

2 P2P-Based Directories in Multi-agent Systems

2.1 Multi-agent Systems

A multi-agent system (MAS) is a loosely coupled network of software agents that cooperate to solve problems that may be beyond the individual capacities or knowledge of each particular agent. In a MAS, computational resources are distributed across a network of interconnected agents. When compared to a centralized system, a MAS does not suffer from the single point of failure problem. Furthermore, a MAS has less performance bottlenecks or resource limitations. Finally, MAS efficiently retrieves, filters, and globally coordinates information from sources that are spatially distributed.

To fulfill their goals, agents in a MAS need to use resources provided by other agents. To use a resource, an agent must contact the agent that provides it. However, in an *open* MAS, an agent does not know which agents provide which resources. Furthermore, it does not know which other agents participate in the system. A common approach to the resource discovery problem is to introduce middle agents or directories that maintain information about which agents provide which resources. Thus to find a resource, an agent has first to contact the middle agent.

However, middle agents can become bottlenecks and contradict the distribution goals set by a MAS along the dimensions of computational efficiency, reliability, extensibility, robustness, maintainability, responsiveness, and flexibility.

2.2 Peer-to-Peer Systems

Recently, peer-to-peer (p2p) computing [8] has evolved as a new distributed computing paradigm of sharing resources available at the edges of the network. A p2p system is a fully-distributed cooperative network in which nodes collectively form a system without any supervision. P2p systems offer robustness in failures, extensive resource sharing, self-organization, load balancing and anonymity.

An issue central to p2p systems is discovering a peer that offers a particular resource. There are two types of p2p systems depending on the way resources are located in the network. In *structured p2p systems*, resources are not placed at random peers but at peers at specified locations. Most resource discovery procedures in structured p2p systems (such as CAN [9], Chord [13], Past [10] and Tapestry [15]) build a distributed hash table. With distributed hashing, each resource is associated with a key and each node (peer) is assigned a range of keys. In *unstructured p2p*, resources are located at random points. In this context, flooding-based approaches to resource discovery have been proposed, in which each peer searching for a resource contacts all peers in its neighborhood. Gnutella [4] is an example of such an approach.

In this paper, we apply fully-decentralized, unstructured p2p search approaches to the problem of resource discovery in open agent systems. Such approaches distribute the load, increase tolerance to failures, are extensible and scalable to the number of agents and resources.

2.3 Distributed Caches

We assume a multi-agent system with N nodes/agents, where each agent provides a number of resources. We assume that there are R different types of resources. Each agent can locally store part of what a middle agent knows. In

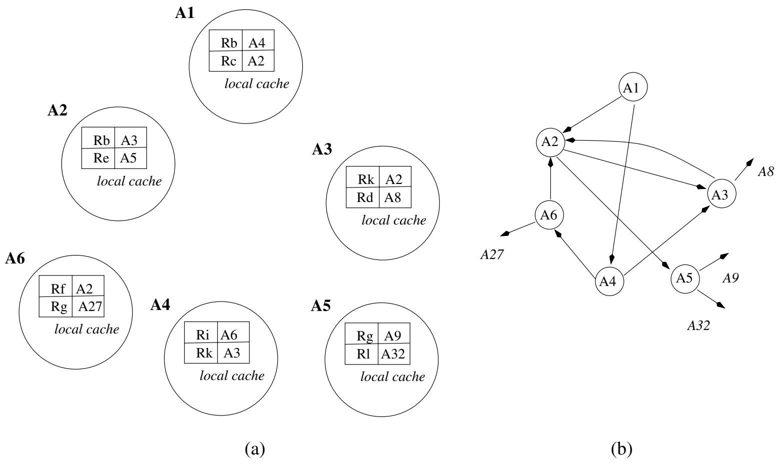

Fig. 1. Part of the cache network: (a) agents with their caches, (cache entries are of the form (Rx, Ai), where Rx is a resource (type) and Ai is the contact information of an agent that offers the resource Rx), and (b) the corresponding directed graph (the cache network).

particular, we assume that each agent has a private cache of size k. Each agent A stores in its cache information about k different resources, that is, for each of the k resources the contact information of one agent that provides it. The agents in the cache of agent A are called A's *neighbors*.

The system is modeled as a directed graph $G(V, E)$, called the *cache network*. Each node corresponds to an agent along with its cache. There is an edge from node A to each of A's neighbors. There is no knowledge about the size of V or E. An example is shown in Figure 1. An agent may provide two or more resources, thus the same agent may appear more than once in another agent's cache. Consequently, there may be less that k outgoing edges from a node, i.e. a node has *at most* k neighbors.

We address the following problem: Given this cache network, how can an agent A that needs a particular type of resource x, find an agent that provides it? Agent A initially searches its own cache. If it finds the resource there, it extracts the corresponding contact information and the search ends. If resource x is not found in the local cache, A sends a message querying a *subset* of its neighbors, which in turn propagate the message to a subset of their neighbors and so on.

Due to the possibility of non-termination, we limit the search to a maximum number of steps, t (similar to the Time To Live (TTL) parameter in p2p systems). In particular, the search message contains a counter field initialized to t. Any intermediate agent that receives the message first decrements the counter by 1. If the counter value is not 0, the agent proceeds as normal; while if the counter value is 0 the agent does not contact its neighbors and sends a positive (negative) response to the inquiring agent if x is found (not found) in its cache.

When the search ends, the inquiring agent A will either have the contact information for resource x or a set of negative responses. In the latter case, agent A assumes that the cache network is *disconnected*, i.e., that it cannot locate x through the cache network. In this case, it will have to resort to other methods, e.g. use a middle agent. Note that disconnectedness may indeed occur because the network is dynamic: caches evolve over time and agents enter and leave the system dynamically.

3 Resource Discovery

We propose three different search strategies based on which subset of its neighbors each agent contacts, namely the flooding, teeming and random paths search strategies. We introduce first our analytical performance model and then present the three search strategies along with an evaluation of their performance.

3.1 Our Perfomance Model

We evaluate the performance of each search strategy with respect to the following three metrics.

- The probability, Q_t, that the resource is found within the t steps. This probability determines the frequency with which an agent avoids using the other locating mechanisms available.
- The average number of steps, $\overline{S_t}$, needed for locating a resource (given that the resource is found).
- The average number of message transmissions, $\overline{M_t}$, occurring during the course of the search. Efficient search strategies should require as few messages as possible in order to not saturate the resources of the underlying network (which, however, may lead to a higher number of steps).

Table 1 summarizes the notation used.

The network of caches is assumed to be in steady-state, all caches being full, meaning that each node knows of exactly k resources (along with the agents that provide them).

If s_i is the probability of locating a resource x at exactly the ith step of a search strategy, then the probability of locating x in any step (up to a maximum of t steps) is given by:

$$Q_t = \sum_{i=0}^{t} s_i. \tag{1}$$

Given that a resource is located within t steps, the probability that we locate it at the ith step is given by s_i/Q_t, and the average number of steps is given by:

$$\overline{S_t} = \sum_{i=1}^{t} i\frac{s_i}{Q_t} = \frac{1}{Q_t}\sum_{i=1}^{t} is_i. \tag{2}$$

Table 1. Notation used.

Input Parameters	
R	number of resource types
N	number of agents (caches)
k	cache size per agent/node
t	maximum allowable number of steps
h	fraction of caches at which a hot spot appears
r_h	percentage of the resources that are hot spots
Output Parameters	
$PC(j)$	probability that a particular resource is in at least one of j given caches
Q_t	prob. of locating a resource within t steps
$\overline{S_t}$	average # steps needed to locate a resource
$\overline{M_t}$	average # of message transmissions
a	$= 1 - k/R$ (random resource requests) $= 1 - h$ (search for a hot resource) $= 1 - \frac{(k/R)-hr_h}{1-r_h}$, (search for a cold resource)

Let $PC(1)$ be the probability of finding x in any given cache. Then, the probability of finding x is in at least one of j given caches is:

$$PC(j) = 1 - (1 - PC(1))^j \tag{3}$$

that is, 1 minus the probability of not finding x in any of the j caches.

Next, we compute $PC(1)$ for two types of resource request distributions.

Uniformly Random Requests. The content of each cache is assumed to be completely random; in other words the cache's k known resources are a uniformly random subset of the R available resources.

Given a resource x, the probability that x is present in a particular cache is equal to:

$$PC(1) = P[x \in \text{cache}] = 1 - P[\text{every cache entry} \neq x].$$

The number of ways to choose k elements out of a set of R elements so that a particular element is not chosen is $\binom{R-1}{k}$. Since the k elements of the cache are chosen completely randomly, the last probability above is equal to: $\binom{R-1}{k}/\binom{R}{k} = (R-k)/R$, which, gives $PC(1) = k/R$. In what follows, we let $a = 1 - k/R$, so that $PC(1) = 1 - a$.

Requests with Hot Spots. In practice, some resources *(hot spots)* are needed more frequently than others *(cold spots)*. In such cases, it is expected that hot spots will appear in a large number of caches. In particular, we assume that a hot spot will appear in a (high) fraction h of all caches, $h \leq 1$. Let a portion r_h of the resources be hot spots, i.e., a total of $r_h R$ resources are hot spots (with the remaining $(1 - r_h)R$ being cold spots).

Searching for hot spots: The probability of finding a particular hot spot in a particular cache is:

$$PC(1) = h = 1 - a \quad \text{where } a \text{ in this case is equal to } 1 - h.$$

Searching for cold spots: We now estimate the probability $PC(1)$ of finding a particular cold spot in a particular cache. Given N agents (caches), each one holding contact information for k resources, there are in total kN cache entries in the whole network. Each hot spot appears in hN caches, and thus $r_h RhN$ entries are occupied by hot spots, in total. The rest will be occupied by cold spots, which means that each of the cold resources appears on the average:

$$\frac{kN - r_h RhN}{R - r_h R} = \frac{N(k - r_h Rh)}{R(1 - r_h)}$$

times, or equivalently, in a portion of

$$\frac{N(k - r_h Rh)}{R(1 - r_h)} \Big/ N$$

of the caches. Consequently, we obtain:

$$PC(1) = \frac{k - hr_h R}{R(1 - r_h)} = \frac{(k/R) - hr_h}{1 - r_h}.$$

Again, we set $a = 1 - \frac{(k/R) - hr_h}{1 - r_h}$, so that $PC(1)$ becomes equal to $1 - a$.

3.2 Flooding

In flooding, agent A that searches for a resource x checks its cache, and if the resource is not found there, A contacts *all* its neighbors (i.e., all the agents listed in its cache). In turn, A's neighbors check their caches and if the resource is not found locally, they propagate the search message to *all* their neighbors. The procedure ends when either the resource is found or a maximum of t steps is reached. The scheme, in essence, broadcasts the inquiring message.

As the search progresses, a k-ary tree is unfolded rooted at the inquiring node A. An example is shown in Figure 2(a). The term "tree" is not accurate in graph-theoretic terms since a node may be contacted by two or more other nodes but we will use it here as it helps to visualize the situation. This search tree has (at most) k^i different nodes in the ith level, $i \geq 0$, which means that at the ith step of the search algorithm there will be (at most) k^i different caches contacted. Since an agent A may offer more than one resource, it may appear more than once in another node's cache. Also, there may exist more than one

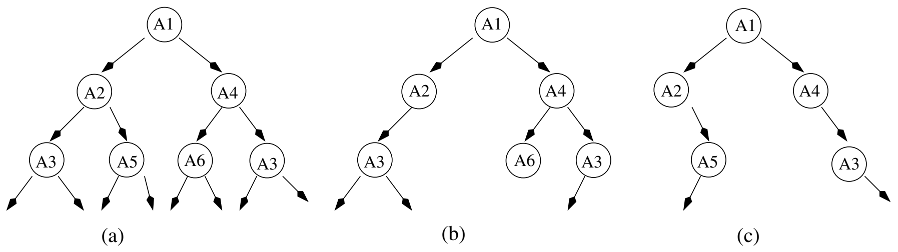

Fig. 2. Searching the cache network of Figure 1: (a) flooding, (b) teeming, (c) random paths ($p = 2$)

caches that know of A. Both these facts may limit the number of different nodes in the ith level of the tree to less than k^i.

Suppose that we are searching at the ith level of this tree for a particular resource x. The probability that we find it there is approximately given by $\ell_i = PC(k^i)$ since in the ith level there are k^i caches. The approximation overestimates the probability since, as noted above, the number of different caches may be less than k^i. However, it simplifies the analysis and does not introduce significant error as shown by our simulation results. Table 2 shows the three performance metrics for flooding. Details for their derivation can be found in [2]. As anticipated, the flooding algorithm requires an exponential number of messages with respect to the cache size (k).

Table 2. Performance of flooding (a is given in Table 1).

Q_t	$1 - a^{\frac{k^{t+1}-1}{k-1}}$
$\overline{S_t}$	$\frac{1}{Q_t}\left(a - (t+1)(1-Q_t) + \sum_{i=2}^{t+1} a^{\frac{k^i-1}{k-1}}\right)$
$\overline{M_t}$	$c^t + c + c(c+1-a)\frac{c^{t-1}-1}{c-1}, \qquad c = ak.$

Figure 3 shows the performance of flooding with respect to the maximum number of steps. The flooding scheme has a number of disadvantages. One is the excessive number of messages that have to be transmitted, especially if t is not small. Another drawback is the way disconnectedness is determined. The inquiring agent has to wait for all possible answers before deciding that it cannot locate the resource. This introduces a number of problems. There is a large number of negative replies. Furthermore, since the network is not synchronized, messages propagate with unspecified delays. This means that the reply of one or more nodes at the tth level of the tree may take quite a long time. One solution is the use of timeout functions; at the end of the timeout period the inquiring

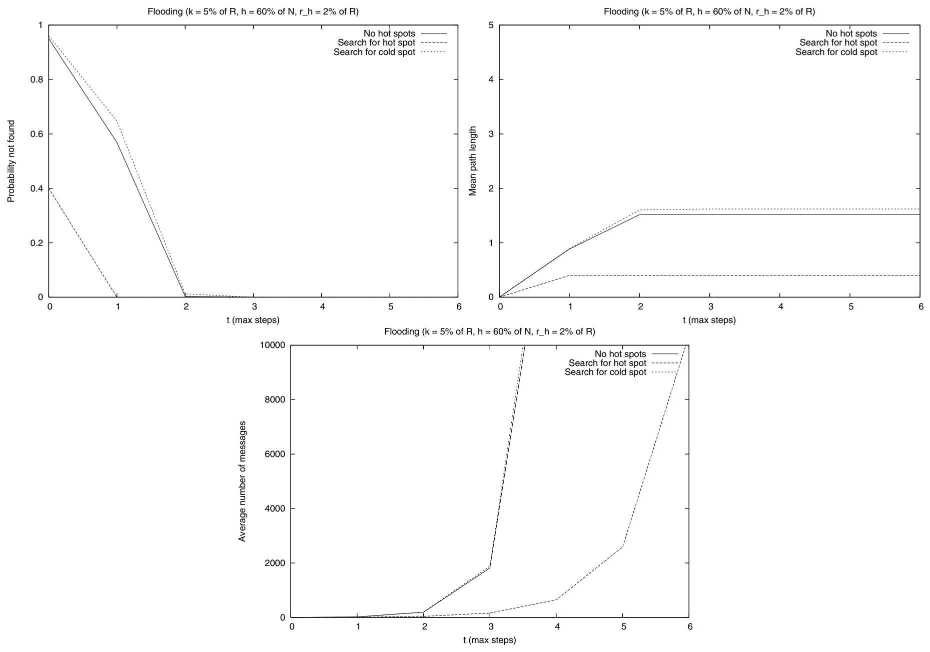

Fig. 3. Performance of flooding (for random distribution (no hot spots), and for hot and cold spots in the case of skewed distribution: probability of not finding the resource ($= 1 - Q_t$), mean path length ($= \overline{S_t}$) and average number of message transmissions ($= \overline{M_t}$).

agent A decides that the resource cannot be located, even if it has not received all answers.

3.3 Teeming

To reduce the number of messages, we propose a variation of flooding that we call *teeming*. With teeming, at each step, if the resource is not found in the local cache of a node, the node propagates the inquiring message only to a *random subset* of its neighbors. We denote by ϕ the fixed probability of selecting a particular neighbor. In contrast with flooding, the search tree is not a k-ary one any more (Figure 2(b)). A node in the search tree may have between 0 to k children, $k\phi$ being the average case. Flooding can be seen as a special case of teeming for which $\phi = 1$.

In teeming, a node propagates the inquiring message to each of its neighbors with a fixed probability ϕ. If the requested resource x is not found, it is due to two facts. First, the inquiring node does not contain it in its cache (occurring with probability $1 - PC(1)$). Second, none of the k "subtrees" unfolding from the inquiring node's neighbors replies with a positive answer. Such a subtree has

$t-1$ levels; it sends an affirmative reply only if it asked by the inquiring node and indeed locates the requested resource.

Based on the above observations, one can derive the three performance metrics of teeming shown in Table 3 (their detailed derivation can be found in [2]). Teeming also requires an exponential number of messages, which however grows slower than in the case of flooding; its rate is controlled by the probability ϕ.

Table 3. Performance of teeming (a is given in Table 1).

Q_t	$1 - a\,(1 - \phi Q_{t-1})^k$
$\overline{S_t}$	$t - \frac{1}{Q_t}\sum_{i=0}^{t-1} Q_i$
$\overline{M_t}$	$c^t + c + c(c+1-a)\frac{c^{t-1}-1}{c-1}$, $c = ak\phi$

3.4 Random Paths

Although, depending on ϕ, teeming can reduce the overall number of messages, it still suffers from the rest of the problems of flooding. One approach to eliminate these drawbacks is the following: each node contacts only one of its neighbors (randomly). The search space formed ends up being a single random path in the network of caches. This scheme propagates one single message along the path and the inquiring agent will be expecting one single answer.

We generalize this scheme as follows: the root node (i.e., the inquiring agent A) constructs $p \geq 1$ random paths. In particular, if x is not in its cache, A asks p out of its k neighbors (not just one of them). All the other (intermediate) agents construct a simple path as above, by asking (randomly) exactly one of their neighbors. This way, we end up with p different paths unfolding concurrently (Figure 2(c)). The search algorithm produces less messages than flooding or teeming but needs more steps to locate a resource.

When using the random paths algorithm, the inquiring node transmits the message to $p \geq 1$ of its neighbors. Each neighbor then becomes the root of a randomly unfolding path. There is a chance that those p paths meet at some node(s), thus they may not always be disjoint. However, for simplification purposes we assume that they are completely disjoint and thus statistically independent. This approximation introduces negligible error (especially if p is not large) as our experiments showed.

At each step i, $i > 0$, of the algorithm, p different caches are contacted (one in each of the paths). The probability of finding resource x in those caches is $PC(p) = 1 - a^p$. One can then derive the three performance metrics for the random paths search strategy shown in Table 4. Details for their derivation can be found in [2].

Table 4. Performance of random paths (a is given in Table 1).

Q_t	$1 - a^{pt+1}$
$\overline{S_t}$	$\frac{a-(1+t-ta^p)(1-Q_t)}{(1-a^p)Q_t}$
$\overline{M_t}$	$ap + ap\frac{1-a^t}{1-a}$

3.5 Performance Comparison

The three performance measures are shown in Figure 4 for all the proposed strategies. In the plots we have assumed cache sizes equal to 5% of the total number of resources R, which was taken equal to 200. The graphs show the random paths strategy for $p = 4$ paths. For the teeming algorithm, we chose $\phi = 1/\sqrt{k}$, that is, on the average $\sqrt{k}$ children receive the message each time. Larger values of ϕ will yield less steps but more message transmissions.

Note that the flooding and teeming algorithms depend on k (the cache size) while the random paths algorithm is only dependent on the ratio k/R (as can be seen by Table 4 for the values of a given in Table 1). The reported results are for searching hot spots. They suggest that caching hot spots is very efficient since for all approaches 2 steps suffice to locate them with very high probability. Although teeming as compared to random paths, yields higher probabilities of locating the requested resource and with a smaller number of steps, the number of message transmissions is excessive. This is because even if the resource is found, some agents may continue to propagate the request, since they are not informed about the successful location of the resource by some other agent.

To validate the theoretical analysis, we developed simulators for each of the proposed strategies. Our simulation results show that our analysis matches the simulation results closely; the approximations made produce negligible error which only shows up in cases of small cache sizes. Figure 5 reports both the analytical estimation and the simulation results for the average number of messages metric. A more detailed discussion can be found in [2].

4 Updates and Mobility

Open MASs are by nature dynamic systems. Agents may move freely, they may offer additional resources or may cease offering some resources. In effect, this gives rise to two types of updates that can make the cache entries obsolete: (i) updates of the contact details (i.e., location) of an agent (for example, when the agent is mobile and moves to a new network site), or (ii) updates of the resources offered by an agent. Note that the second type of updates models also the cases in which an existing agent leaves the MAS (cease to offer all its resources) or a new agent joins the MAS (offers additional resources).

An approach to handling updates due to agent mobility is to change the type of cache entries. In particular, instead of storing in the cache the location

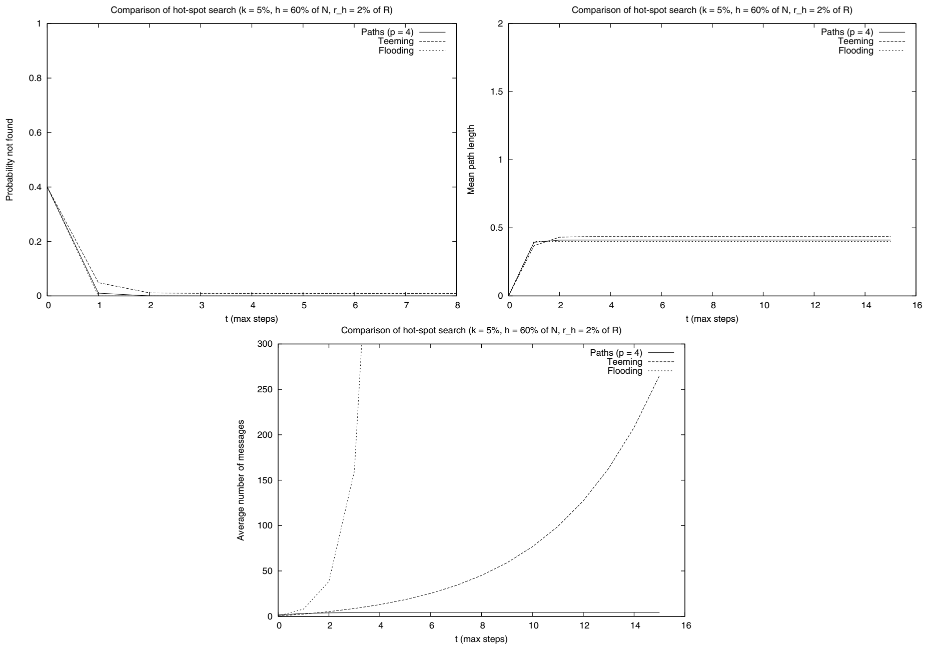

Fig. 4. Comparison of the proposed algorithms: probability of not finding the resource ($= 1 - Q_t$), mean path length ($= \overline{S_t}$) and average number of message transmissions ($= \overline{M_t}$). The teeming algorithm uses $\phi = 1/\sqrt{k}$. The resource searched for is a hot spot.

of an agent, we may maintain just its name. An additional location server is then needed that maintains a mapping between agent names and their locations. In this case, updates of an agent's location do not affect any of the caches. However, this approach adds the additional overhead of contacting the location server, which can now become a bottleneck. In this paper, we consider only decentralized approaches. We also focus on location updates; similar considerations hold for resource updates as well.

Any cache updates are initiated from the agent that moves. The agent may either send an invalidation update message or a propagation update message. In the *invalidation* case, the agent just sends a message indicating the update, so that the associated entries in the caches are marked invalid. In the *propagation* case, the agent also sends its new location. In this case, the associated cache entries are updated with the new location. Invalidation messages are smaller than propagation messages and work well with frequent moving agents. A hybrid approach is also possible. For instance, a frequent moving agent sends an invalidation message first, and a propagation message containing its new location later after settling down at a location.

Next, we consider two approaches to cache invalidation: one based on the notion of an inverted cache and one based on flooding.

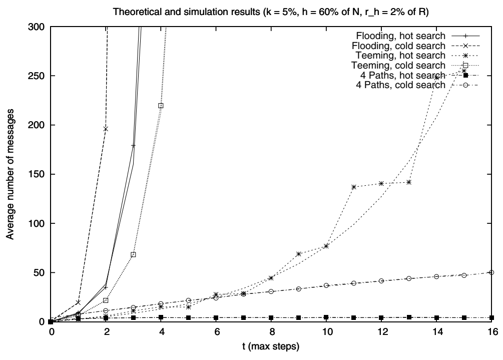

Fig. 5. Validation of our analytical models.

4.1 Inverted Cache

One approach is to maintain an "inverted" cache at each agent. In particular, each agent A maintains a list of all agents that know about A, that is, all agents that have A in their cache. Figure 6 shows an example instance of such a cache.

When an agent moves, it uses the inverted cache to find out which agents it needs to contact. Then, it sends an invalidation (propagation) message to them. In the example of Figure 6, when $A5$ moves, it needs to inform agents $A2$ and $A7$. In terms of the corresponding graph, the dissemination of the updates follows the dotted arrows. Note, that only the agents in the inverted cache need to be contacted.

For "popular" agents, that is, agents with resources that are hot spots, the size of the inverted cache may become very large. Also, the maintenance of an inverted cache makes cache management harder, since each time an entry for a resource offered by B is cached at an agent A, A needs to inform B so that B includes A in its inverted cache. Another consideration is whether the inverted cache should be used in resource discovery: should the agents in an agent's inverted cache be contacted during search?

4.2 Flooding-Based Dissemination of Updates

It is possible to disseminate invalidation (or propagation messages) using an approach similar to the proposed approaches for resource discovery. When an agent A moves, it informs some of its neighbors (i.e., the k agents that are in its cache) by sending an invalidation (propagation) message to them. Each one of them, checks whether agent A is in their cache, and if so it invalidates the corresponding entry (or updates it with A's new location). Then, it forwards the message to some of its neighbors. This process continues until a maximum number of steps is reached. Based on which subset of its neighbors an agent selects to inform at each step, we may have flooding, teeming or random path variations of this procedure.

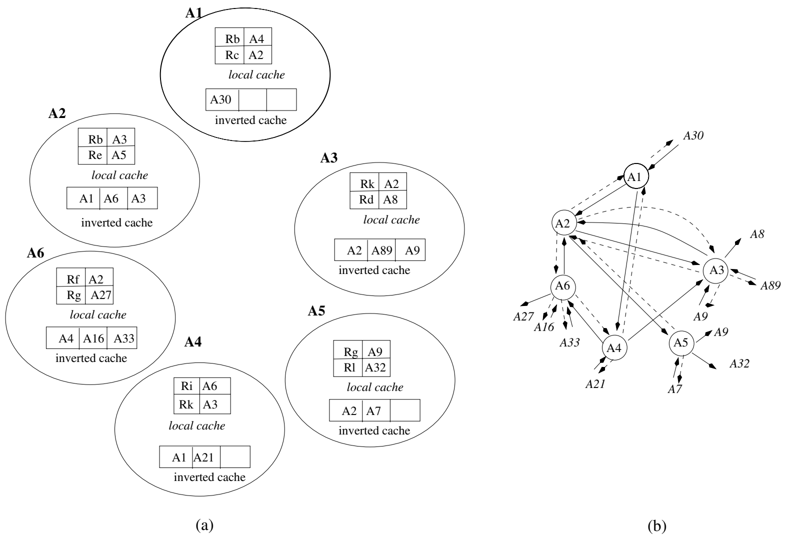

Fig. 6. The extended cache: (a) agents, (b) corresponding graph (dotted edges correspond to inverted cache entries)

How many cache entries will be informed depends on the maximum number of steps. It also depends on the topology of the cache network, since in our model, the neighbor relationship is not symmetric; that is, B may be a neighbor of A (i.e., B may be in A's cache), while A is not a neighbor of B (i.e., A is not in B's cache). For example, this is the case when A needs resources offered by B, while B does not need any of A's resources. With flooding-based dissemination, some entries may still be obsolete. In this case, an agent discovers this fact when attempting to contact an agent using an outdated location. The agent may invalidate the entry and continue the search.

5 Related Work

The only other studies of the use of local caches for resource location in MAS that we are aware of are [1] and [12]. In [1], a depth first traversal of what corresponds to our cache network is proposed. Experimental results are presented that show that this approach is more efficient in terms of the number of messages than flooding for particular topologies, in particular, the ring, star, and complete graph topologies. There are no analytical results. In [12], the complexity of the very limited case of lattice-like graphs (in which each agent knows exactly four other agents in such a way that a static grid is formed) is analyzed.

The problem that we study in this paper can be seen as a variation of the resource discovery problem in distributed networks [6], where nodes learn about other nodes in the network. However, there are important differences: (i) we are interested in learning about one specific resource as opposed to learning about all other known nodes, (ii) our network may be disconnected and (iii) in our case, each node has a limited-size cache, so at each instance, it knows about at most k other nodes. A similar problem appears also in resource discovery in peer-to-peer (p2p) systems. While there have been a lot of empirical studies (e.g. [11]) and some simulation-based analysis (e.g. [7]) of flooding and its variants for p2p systems, analytical results are lacking. Here, we analytically evaluate various alternatives of flooding-based approaches. Finally, flooding has also been used in ad-hoc routing (e.g. [5]). Here, the objective is to ensure that a message starting from a source node reaches its destination.

The cache network for agents was first introduced in [3]. In this paper, we extend our analysis for a skewed distribution and obtain performance results for discovering hot and cold spots. In addition, we consider the problem of cache updates.

6 Conclusions

In this paper, we focused on resource location in multi-agent systems. We proposed a fully distributed approach, in which each agent maintains in a local cache information about a number of resources. We introduce and analytically estimated the performance of a number of variations of flooding-based search using these caches for both a random and a skewed distribution. A problem that was not addressed in this paper is how to choose which resources to cache at each agent. This is an interesting problem that we are currently pursuing. We are also implementing a prototype of our system in a mobile agent platform.

Acknowledgments. This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-32645 DBGlobe project and by a Greek Ministry of Education program for Supporting Graduate Studies in Computer Science (EPEAEK II).

References

1. M. A. Bauer and T. Wang. Strategies for Distributed Search. In *CSC '92, ACM Conference on Computer Science*, 1992.
2. V. V. Dimakopoulos and E. Pitoura. A Peer-to-Peer Approach to Resource Discovery in Multi-Agent Systems (Extended Vesrion). Technical Report TR2003, Univ. of Ioannina, Dept. of Computer Science, June 2003. Also available at: http://www.cs.uoi.gr/~ pitoura/pub.html.
3. V. V. Dimakopoulos and E. Pitoura. Performance Analysis of Distributed Search in Open Agent System. In *IPDPS '03, International Parallel and Distributed Processing Symposium*, 2003.
4. Gnutella website. http://gnutella.wego.com. Technical report.
5. Z. Haas, J. Y. Halpern, and L. Li. Gossip-Based Ad Hoc Routing. In *IEEE Proc. of INFOCOM 2002*, pages 1707–1716, 2002.

6. M. Harchol-Balter, T. Leighton, and D. Lewin. Resource Discovery in Distributed Networks. In *PODC '99, Principles of Distributed Computing*, pages 229–337, 1999.
7. Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. ICS2002, 16th ACM Int'l Conf. on Supercomputing*, pages 84–95, 2002.
8. D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL2002-57, HP Technical Report, 2002.
9. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-Addressable Network. In *Proc. of ACM SIGCOMM*, pages 161–172, 2001.
10. A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of SOSP 2001, 18th ACM Symp. on Operating System Priciples*, pages 188–201, 2001.
11. S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Proc. MMCN '02, Multimedia Computing and Networking 2002*, 2002.
12. O. Shehory. A scalable agent location mechanism. In *Proc. ATAL '99, 6th Int'l Workshop on Intelligent Agents, Agent Theories, Architectures, and Languages*, volume 1757 of *LNCS*, pages 162–172. Springer, 2000.
13. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proc. of ACM SIGCOMM*, pages 149–160, 2001.
14. K. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic Service Matchmaking Among Agents in Open Information Environments. *SIGMOD Record*, 28(1):47–53, March 1999.
15. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, U. C. Berkeley, April 2001.