A Peer-to-Peer Aproach to Resource Discovery in Mulit-Agent Systems (extended version)

Vassilios V. Dimakopoulos and Evaggelia Pitoura

Technical Report 2003-09

Department of Computer Science, University of Ioannina
P.O. Box 1186, Ioannina, Greece GR-45110
Tel: +30 26510 {98809, 98811}, Fax: +30 26510 98890
E-mail: {dimako, pitoura}@cs.uoi.gr

June 2003

Abstract

In open multi-agent systems, agents need resources provided by other agents, but they are not aware of which agents provide the particular resources. Most solutions to this problem are based on a central directory that maintains a mapping between agents and resources. However, such solutions do not scale well, since the central directory becomes a bottleneck in terms of both performance and reliability. In this paper, we introduce a different approach: each agent maintains a limited size local cache in which it keeps information about k different resources, that is, for each of k resources, it stores the contact information of one agent that provides it. This creates a directed network of caches. We address the following fundamental problem: how can an agent that needs a particular resource, find an agent that provides it by navigating through this network of caches. We propose and analytically compare the performance of three different algorithms for this problem, flooding, teeming and random paths, in terms of three performance measures: the probability to locate the resource, and the number of steps and messages to do so.

1. Introduction

In this paper, we consider the following problem. Let $G(V, E)$ be a directed, not necessarily connected graph, where each node $v \in V$ is connected with at most k other nodes. If there is a directed edge from a node v to another node u, we say that v *knows* about u. We address the following questions: starting from an arbitrary node v how can we reach (learn about) another node u, what is the probability to reach u and what is the associated communication cost.

This problem arises in *open* multi-agent systems (MAS), where agents need resources or services provided by other agents. 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 [4]). 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 the bottlenecks of the system both from a performance and from a reliability perspective.

In this paper, we introduce a new approach to the problem of resource locating in agent systems. Each agent is equipped with a limited size local "cache" so as to maintain contact information for up to k different resources (i.e., for each of the k resources, the agent knows an agent that offers it). This results in a fully distributed directory scheme, where each agent stores part of the directory. We model this system as a *network of caches*. There is a link from node v to node u if agent v knows agent u, that is agent's u contact information is stored in v's cache.

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.

There are a number of problems that have to be solved in order to make such a mechanism work properly and efficiently. For one, the system should be adaptive in many different ways: cached data change over time, giving rise to cache replacement policies; agent locations may also do so, possibly invalidating cache entries.

In this work we mainly address a fundamental issue: *resource location*. The general mechanism for locating a resource is as follows: The agent that requires a resource first looks at its cache. If no contact information for the resource is found in the cache, the agent selects other agent(s) from its cache, contacts them and inquires their local cache for the resource. This procedure continues until either the resource is located or a maximum number of steps is reached. In essence, this procedure constructs directed path(s) in the network of caches. If the resource cannot be found, the agent has to resort inevitably to some other (costly) mechanism (e.g. to a middle agent) which is guaranteed to reply with the needed information.

For the resource location problem, we provide a number of different solutions that adhere to the aforementioned procedure. In particular, we study the effectiveness of using single and multiple search paths. We also provide performance analysis results. The performance metrics we are interested in are:

- the probability to locate a resource within t steps,
- the average number of steps needed to locate a resource and
- the average number of message transmissions required.

The probability of locating a resource is important because it is directly connected with the frequency with which an agent resorts the the 'other' mechanisms (e.g. middle agents), and should be as high as possible. On the other hand, the number of message transmissions which is but one measure of the communication cost, should be as low as possible. For each of the above quantities, we provide analytical estimations and simulation results that verify them.

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.

1.1. The model

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 type of resources. To fulfill their goals, agents need to use resources provided by other agents. To use a resource, an agent must contact the agent that provides it. However, an agent does not know which agents provide which resources. Furthermore, it does not know which other agents participate in the system. Thus to find a resource, an agent has first to contact a middle agent.

The system is modeled as a directed graph $G(V, E)$, called the *caches network*. Each node corresponds to an agent along with its cache. There is an edge from a node v to a node u if and only if agent v has in its cache the contact information of agent u. There is no knowledge about the size of V or E. An example is shown in Fig. 1.

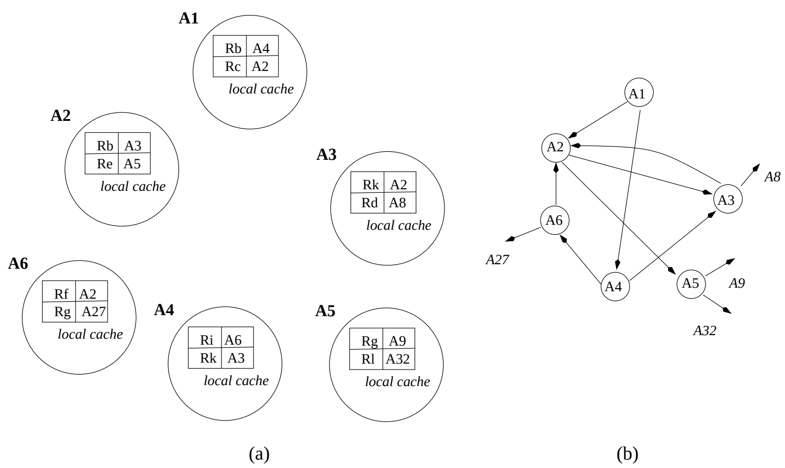

Figure 1. *Part of a cache network, each agent v_i maintains in its cache the contact information for two resources ($k = 2$)*

1.2. Related work

The only other study of the use of local caches for resource location in open MAS that we are aware of is that in [3]. However, in this work, only 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.

Searching in a network model closely resmbling ours was also considered in [1]. The authors proposed depth-first search strategies in order to minimize the number of messages. They studied the algorithms for particular topologies, namely rings, stars and complete graphs, but no analytic results were derived.

The problem can be seen as a variation of the resource discovery problem in distributed networks [2], 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.

1.3. Organization

The remainder of this paper is structured as follows. Section 2 presents a series of algorithms for the resource location problem. In Section 3 we present our analytical model and study analytically the performance of the candidate algorithms. In Section 4 we describe the simulator we constructed along with experimental results, which validate our analytical models. In Section 5 we presents our thoughts on the architectural details of the prposed system that are related to agent mobility. Finally, in Section 6 we discuss our plans for future work.

2. Candidate algorithms

Consider an agent A, called the *inquiring agent*, which needs a particular type of resource (or service) x. In order to obtain this resource, it must find another 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 the agents found in its cache, that is, some of A's neighbors, which in turn propagate the message to query 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. In particular, the inquiring message contains a counter field initialized to t. Any intermediate node that receives the message first decrements the counter by 1. If the counter value is not 0, the agent proceeds as normal; if, on the other hand, 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 answers. In the latter case, agent A assumes that the cache graph is *disconnected* i.e., that it cannot locate x through the cache graph. 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.

In the following sections we propose three different strategies for choosing the subset of neighbors each node contacts.

2.1. Flooding

In flooding, A contacts *all* its neighbors (i.e., all the agents listed in its cache), by sending an inquiring message, asking for information about resource x. Any agent that receives this message searches its own cache. If x is found in there, a reply containing the contact information is sent back to the inquiring agent. Otherwise, the intermediate agent contacts all of its own neighbors (agents in its cache), thus propagating the inquiring message. The scheme, in essence, broadcasts the inquiring message.

It is not difficult to see that this scheme floods the network with messages until the particular resource x is found somewhere. As messages are sent from node to node, a "tree" is unfolded rooted at the inquiring agent (see Fig. 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 is as it helps to visualize the situation.

The flooding scheme has a number of disadvantages. One of them is the excessive number of messages that have to be transmitted, especially if t is not small. Another

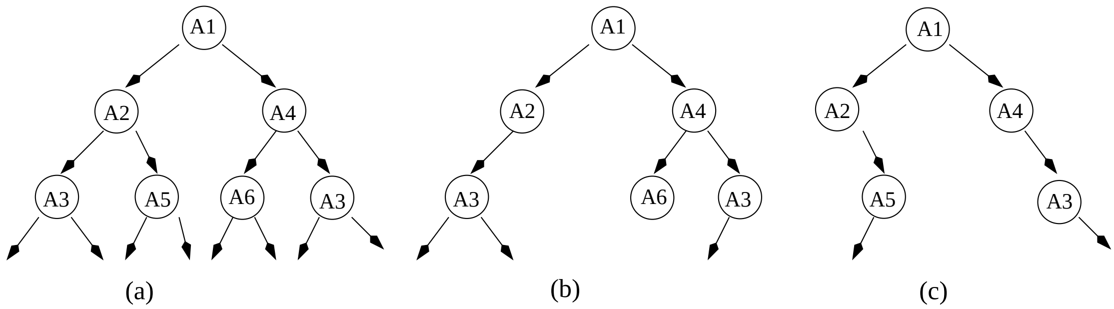

Figure 2. *Searching the cache network of Fig. 1: (a) flooding, (b) teeming, (c) random path ($p = 2$)*

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

Due to the above problems the flooding scheme is probably impractical even for relatively small values of t. We consider next a generalization of flooding, called teeming.

2.2. Teeming

To reduce the number of messages, we propose a new algorithm, termed *teeming*. At each step of this algorithm, 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 neighbor. Let ϕ be the fixed probability of selecting a particular neighbor. In contrast with flooding, the search tree is not a k-ary one any more (Fig. 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 $\phi = 1$.

2.3. Random paths

Although, depending on ϕ, teeming can reduce the overall number of messages, it still suffers from the rest of flooding's problems. To avoid both the excessive number of messages and the delay, we may consider a restricted form of search: each node contacts only one of its neighbors (randomly). We, thus, end up following 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.

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

3. Performance analysis

In this section we analyze the performance of the algorithms presented in the previous section. In particular, we will assume that the algorithms are allowed to operate for a maximum of t steps and we will derive analytically three important performance measures:

- The probability, Q_t, that the resource we are looking for is found within the t steps. This probability determines the frequency with which an agent uses other locating services available; Q_t should be as high as possible.
- The average number of steps, $\overline{S_t}$, needed for locating a resource (given that the resource is found), which naturally should be kept quite low.
- The average number of message transmissions, $\overline{M_t}$, occurring during the course of the algorithm. Efficient strategies should require as few messages as possible in order not to saturate the underlying network's resources (which, however, leads to a higher number of steps).

3.1. Preliminaries

Here is a summary of the notation we will use:

R	number of resource types
k	cache size per agent/node
a	$= 1 - PC(1)$, and depends on cache contents (see Eq. (4))
$PC(j)$	probability that a particular resource is in at least one of j given caches
t	maximum allowable number of steps
s_i	probability of locating a resource in exactly i steps
Q_t	probability of locating a resource within t steps
$\overline{S_t}$	average number of steps needed to locate a resource
$\overline{M_t}$	average number of message transmissions.

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 their providers).

Given a resource type x, assume that the probability that x is present in a particular cache is equal to $PC(1)$. If we are given j such caches, the probability that x is in at least one of them is:

$$\begin{aligned} PC(j) &= 1 - P[x \notin \text{any of the } j \text{ caches}] \\ &= 1 - (1 - PC(1))^j. \end{aligned} \tag{1}$$

Now let us denote by s_i the probability of locating x at exactly the ith step of an algorithm. Then the probability of locating x in any step (up to a maximum of t steps) is simply given by:

$$Q_t = \sum_{i=0}^{t} s_i. \tag{2}$$

An important performance measure is the average number of steps needed to find a resource x. 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 clearly given by:

$$\overline{S_t} = \sum_{i=1}^{t} i \frac{s_i}{Q_t} = \frac{1}{Q_t} \sum_{i=1}^{t} i s_i. \tag{3}$$

Notice that in order to derive the performance measures, we need to calculate $PC(1)$ and s_i. $PC(1)$ is dependent only on the cache contents, while s_i depends on the particular search strategy utilized. We compute $PC(1)$ next, and we defer the derivation of s_i for each stategy to the corresponding subsections.

3.2. Cache composition

We now turn our attention to the contents of caches, i.e. the distirbution of resources over the caches in the network.

Uniformly random. In this first case, we will assume that the content of each cache is completely random; in other words the cache's k known resources are a uniformly random subset of the R available resources. The probability of finding a particular resource x in any such cache is given by:

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

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 clearly equal to:

$$\frac{\binom{R-1}{k}}{\binom{R}{k}} = \frac{(R-k)}{R},$$

which, gives $PC(1) = k/R$. In what follows, we let $a = 1 - k/R$, so that $PC(1) = 1 - a$.

Hot spots. In practice, it is noticed that some resources are more popular than others. The former are called *hot spots* and the latter *cold spots*. Under the presence of hot spots, the cache contents are no longer uniformly random – hot spots / resources will be present in a higher percentage of caches than cold spots.

We will assume that within each class (hot or cold) all resources equiprobable. In particular, each hot spot will be assumed to appear in a percentage h of all caches. Finally, we let r_h denote the fraction of resources that are hot – then r_hR is the total number of hot spots, while the remaining $(1 - r_h)R$ are cold.

Searching for hot spots. If the required resource x is a hot spot, then the probability of finding it in any given cache will be equal to:

$$PC(1) = h = 1 - a,$$

where $a = 1 - h$.

Searching for cold spots. We will now estimate the propability of finding a particular cold spot in a given 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, and since all of them are equiprobable, 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 equivallently, in a portion of

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

of the caches. Consequently, we obtain:

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

Set $a = 1 - \frac{(k/R) - hr_h}{1 - r_h}$, and $PC(1)$ becomes equal to $1 - a$.

In conclusion, the probability of locating a resource x in a given cache is given by:

$$PC(1) = 1 - a, \quad \text{where: } a = \begin{cases} 1 - \frac{k}{R} & \text{uniformly random case} \\ 1 - h & \text{searching for hot spots} \\ 1 - \frac{(k/R) - hr_h}{1 - r_h} & \text{searching for cold spots.} \end{cases} \tag{4}$$

3.3. Performance of flooding

In flooding, each node receiving the inquiring message transmits it to all its neighbors (unless the required resource x is contained in its cache). As the algorithm progresses, a k-ary tree is unfolded rooted at the inquiring node. 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 algorithm there will be (at most) k^i different caches contacted.*

Suppose that we are searching the ith level of this tree for a particular resource type 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.

The probability of locating x at exactly the ith step is given by:

$$s_i = \ell_i \prod_{j=0}^{i-1} (1 - \ell_j), \tag{5}$$

that is, we locate it at the ith level and in none of the previous ones. Substituting yields:

$$\begin{aligned} s_i &= PC(k^i) \prod_{j=0}^{i-1} (1 - PC(k^j)) \\ &= \left(1 - a^{k^i}\right) \prod_{j=0}^{i-1} a^{k^j} \\ &= \left(1 - a^{k^i}\right) a^{\sum_{j=0}^{i-1} k^j}, \end{aligned}$$

or,

$$s_i^{(F)} = \left(1 - a^{k^i}\right) a^{\frac{k^i - 1}{k-1}}, \tag{6}$$

where F is used to denote the flooding scheme. Substituting in (2), and after some manipulation (see Appendix A), we obtain:

$$Q_t^{(F)} = 1 - a^{\frac{k^{t+1} - 1}{k-1}} \tag{7}$$

The average number of steps needed to locate a resource can be found by substituting (6) into (3). After some straightforward manipulations (given in Appendix

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

A), the average number of steps is found to be:

$$\overline{S_t^{(F)}} = \frac{1}{Q_t^{(F)}} \left(a - (t+1)(1 - Q_t^{(F)}) + \sum_{i=2}^{t+1} a^{\frac{k^i - 1}{k-1}}. \right) \tag{8}$$

We know of no closed-form formula for the sum in (8).

Let us now compute the number of messages in the flooding algorithm. If the resource is found in the inquiring node's cache there will be no message transmissions. Otherwise, there will be k transmissions to the k neighbors of the root, plus the transmissions internal to each of the k subtrees T rooted at those neighbors. Symbolically, we have:

$$\overline{M_t^{(F)}} = (1 - PC(1))(k + km(t-1)),$$

where $m(t-1)$ are the transmissions occurring within a particular subtree T with $t-1$ levels. For such a subtree T, if x is found in its root node there will be 1 positive reply back to the inquiring node; otherwise, there will be k message transmissions to the children of the root plus the transmissions inside the k subtrees T' (with $t-2$ levels) rooted at the node's children. We are thus led to the following recursion:

$$\begin{aligned} m(t-1) &= 1 \times PC(1) + (k + km(t-2)) \times (1 - PC(1)) \\ &= akm(t-2) + ak + 1 - a, \end{aligned}$$

with a boundary condition of:

$$m(0) = 1, \tag{9}$$

since the last node receiving the message (at the tth step) will always reply to the inquiring node whether it knows x or not. The solution to the above recursion is:

$$m(t-1) = (ak)^{t-1} + \frac{(ak)^{t-1} - 1}{ak - 1}(ak + 1 - a),$$

which gives:

$$\overline{M_t^{(F)}} = c^t + c + c(c + 1 - a)\frac{c^{t-1} - 1}{c - 1}, \qquad c = ak. \tag{10}$$

Eq. (10) shows that (as anticipated) the flooding algorithm requires an exponential number of messages with respect to cache size (k).

3.4. Performance of teeming

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 (with probability ϕ) and indeed locates the requested resource (with probability $Q_{t-1}^{(T)}$). Thus, the probability of not finding x is given by the following recursion:

$$1 - Q_t^{(T)} = (1 - PC(1)) \left(1 - \phi Q_{t-1}^{(T)}\right)^k,$$

which gives:

$$Q_t^{(T)} = 1 - a \left(1 - \phi Q_{t-1}^{(T)}\right)^k. \tag{11}$$

where T is used to denote the teeming algorithm. The boundary condition is, clearly, $Q_0^{(T)} = PC(1) = 1 - a$.

The average number of steps can be found to be (see Appendix B):

$$\overline{S_t^{(T)}} = t - \frac{1}{Q_t^{(T)}} \sum_{i=0}^{t-1} Q_i^{(T)}. \tag{12}$$

The average number of messages is computed almost identically with the flooding case; the only difference is that since a node transfers the message to a particular child with probability ϕ, the average number of steps will be given by:

$$\overline{M_t^{(T)}} = (1 - PC(1))(k\phi + k\phi m(t-1)),$$

where $m(t-1)$ is the transmissions occurring within a particular subtree T with $t - 1$ levels. The recursion (see Section 3.3) takes the form:

$$m(t-1) = 1 \times PC(1) + (k\phi + k\phi m(t-2)) \times (1 - PC(1)),$$

which finally gives:

$$\overline{M_t^{(T)}} = c^t + c + c(c + 1 - a)\frac{c^{t-1} - 1}{c - 1}, \qquad c = ak\phi. \tag{13}$$

Teeming also requires an exponential number of messages, which however grows slower than flooding's case; its rate is controlled by the probability ϕ.

3.5. Performance of the random paths algorithm

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 will 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 asked (one in each of the paths). The probability of finding resource x in those caches is $\ell = PC(p) = 1 - a^p$. Therefore, the probability of finding x after *exactly* i steps is equal to (analogously to (5)):

$$s_i^{(p)} = \begin{cases} 1 - a & i = 0 \\ a(1 - a^p)a^{p(i-1)} & i \geq 1. \end{cases} \tag{14}$$

Given t steps maximum, we can calculate the probability $Q_t^{(p)}$ that what we are looking for is found, using (2) and (14):

$$Q_t^{(p)} = 1 - a^{pt+1}. \tag{15}$$

Similarly, the average number of steps will be given by:

$$\overline{S_t^{(p)}} = \frac{a - (1 + t - ta)(1 - Q_t^{(p)})}{(1 - a^p)Q_t^{(p)}}. \tag{16}$$

The derivation of both formulas is given in Appendix C. Setting $p = 1$ the above formulas gives the corresponding performance measures for the single-path algorithm.

Finally, the number of message transmissions can be calculated using arguments similar to the ones in Section 3.3. If x is not found at the inquiring node's cache, then there will be p message transmissions to p of its children, plus the message transmissions in each of the p paths:

$$\overline{M_t^{(p)}} = (1 - PC(1))(p + pm(t - 1)),$$

where $m(t - 1)$ is the transmissions occurring within a particular path P of $t - 1$ nodes. For such a path P, if x is found in its root node there will be 1 positive reply back to the inquiring node; otherwise, there will be one message transmission to the next node of the path plus the transmissions inside the subpath P' (with $t-2$ nodes) rooted at the next node. We are thus led to the following recursion:

$$\begin{aligned} m(t-1) &= 1 \times PC(1) + (1 + m(t-2)) \times (1 - PC(1)) \\ &= am(t-2) + 1, \end{aligned}$$

where, as in (9), $m(0) = 1$, since the last node receiving the message will always reply to the inquiring node whether it knows x or not. The solution to the above recursion is easily found to be:

$$m(t-1) = \frac{1-a^t}{1-a},$$

which gives:

$$\overline{M_t^{(p)}} = ap + ap\frac{1-a^t}{1-a}. \tag{17}$$

3.6. Comparison

The three performance measures are shown in Fig. 3 for all the proposed strategies.† Two different cache sizes are considered: 5% and 10% of the total number of resources R, which was taken equal to 200. The flooding and teeming algorithms depend on k while the random paths algorithm is only dependent on the ratio k/R.

The graphs show the random paths strategy for $p = 1$, 2 and 4 paths. For the teeming algorithm, we chose $\phi = 1/\sqrt{k}$; on the average $\sqrt{k}$ children receive the message each time. Larger values of ϕ will yield less steps but more message transmissions as is evident from (12), (13).

The plots show vividly the positive and negative aspects of the algorithms. Flooding/teeming yield higher probabilities of locating the requested resource and within a small number of steps, as compared to random paths. However, the number of message transmissions is excessive by any measure. Teeming constitutes possibly the better tradeoff if the probability ϕ is chosen appropriately.

The random paths strategy can be quite poor for very small values of p (e.g. 1 or 2). However, for 4 paths or more, and larger cache sizes, its performance seems the most balanced of all.

†Here we only present results for uniformly random requests.

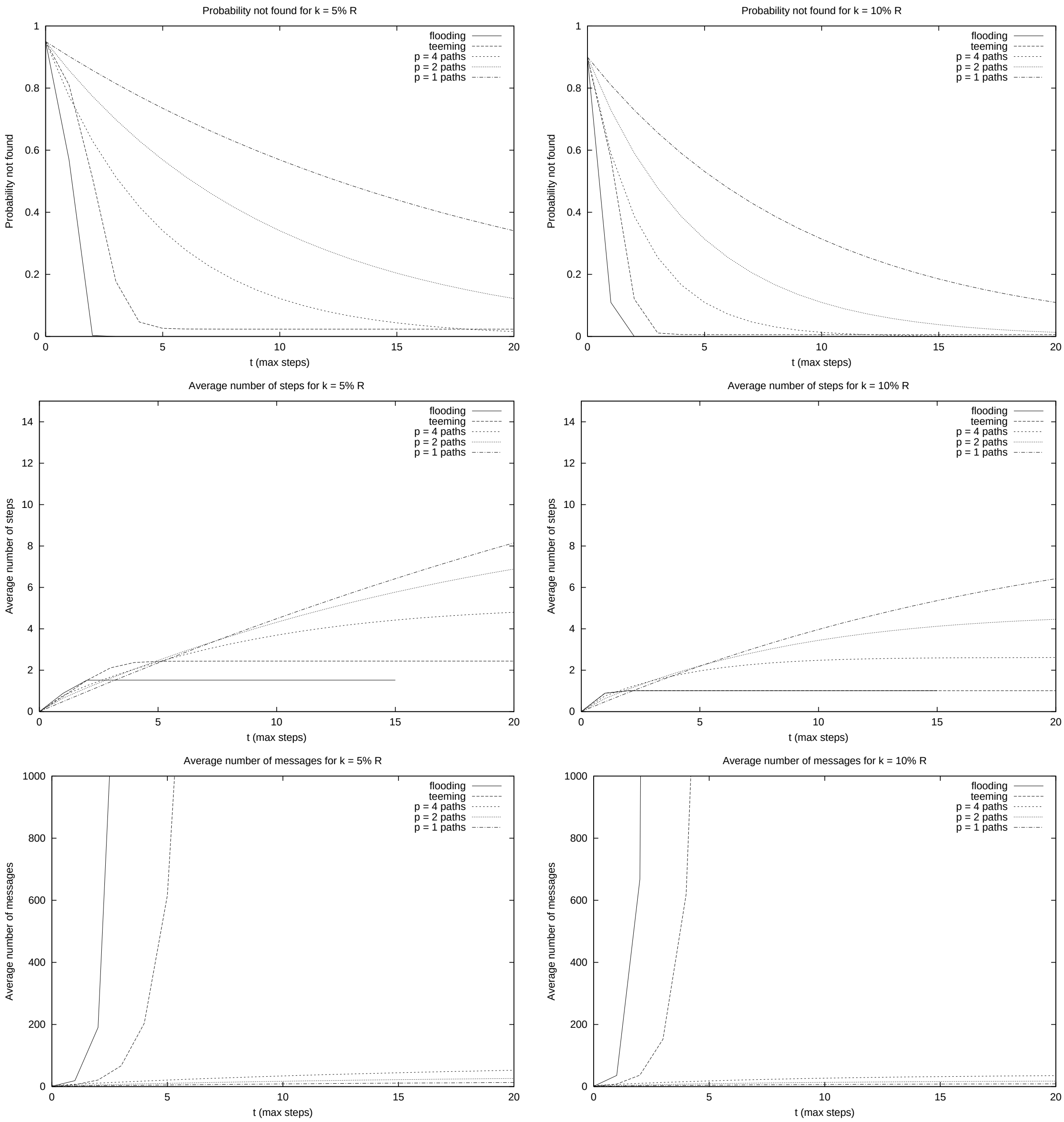

Figure 3. *Comparison of the proposed algorithms: probability of not finding the resource ($= 1 - Q_t$), mean path length ($= \overline{S_t}$) and average number of message transmissions ($= \overline{M_t}$). The p-paths algorithm is shown for $p = 1, 2, 4$. The teeming algorithm uses $\phi = 1/\sqrt{k}$. The results shown here are for $R = 200$ resources and two different cache sizes (k): 5% and 10% of R.*

4. Simulation

To validate the theoretical analysis, we developed simulators (written in C) for each of the proposed strategies. The simulators, upon initialization, construct a table mapping each of the R available resources to random agents (which will provide the resource). There are a total of N agents, where N can be up to 10000.

Next, the caches of all agents are filled with random k-element subsets of the available resources. Notice that those initial cache contents depends on the particular distribution used (see Section 3.2) that is whether there are hot spots or not. The percentage of hot spots (r_h) as well as the percentage of caches that include a particular hot spot (h) are user-defined. Finally, it should be noted that since an agent may provide more than one resource, it may appear more than once in a particular cache; in effect, the degree of a node in the network of caches may be less than k. k can be anywhere from 1 to R.

After the initialization, simulation sessions take place with a random agent issuing one location request for a random resource x each time. In the presence of hot spots, the resource asked for can be either hot or cold. In order to avoid unnecessary loops, when agents select the appropriate neighbors to propagate the inquiring message, they make sure that the node they received the message from is not contacted.

Measures obtained from each simulation session include the number of messages, the path length and a flag denoting whether the resource was found or not. For each of the proposed algorithms, at least 500 such sessions are performed and the accumulated results are averaged.

In Fig. 4 – 5 we give a number of simulation results for each of the proposed algorithms (patterned lines), along with the curves obtained from the theoretical analysis (unpatterned lines). The results include the probability of not locating the resource ($= 1 - Q_t$), the average number of steps ($= \overline{S_t}$) and the average number of message transmissions ($= \overline{M_t}$).

All experiments were run for $N = 2000$ agents and cache size of $K = 10$. For hot-spot simulations, each hot spot was appearing in $h = 60\%$ of the caches. For the teeming strategy the probability ϕ was taken equal to $\sqrt{k}$, while for the random paths strategy we used $p = 4$ paths.

In Fig. 4 the number of resources was $R = 200$ so that the cache size is 5% of R, while $r_h = 2\%$ of the resources were hot-spots. In Fig. 5 we set $R = 1000$ and $r_h = 1\%$, so that the cache size is small as compared to R ($K = 1\% of R$).

It is easily seen that the simulation results match the analytical ones quite closely. The appoximation (that all cache entries contain different agents) in Sections 3.3 and 3.4 produce very small error which shows up only in the case of cold searches and very small cache sizes (Fig. 5); in such cases it can be seen that for cold spots (which occupy few entries in each cache), the probability of not finding the resource and the average number of steps are slightly underestimated. For the random paths algorithm the error introduced by the approximation that all p paths are node-disjoint is negligible even for small caches sizes. There is essentially no error when caches are not too small, for all strategies (Fig. 4).

5. 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 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.

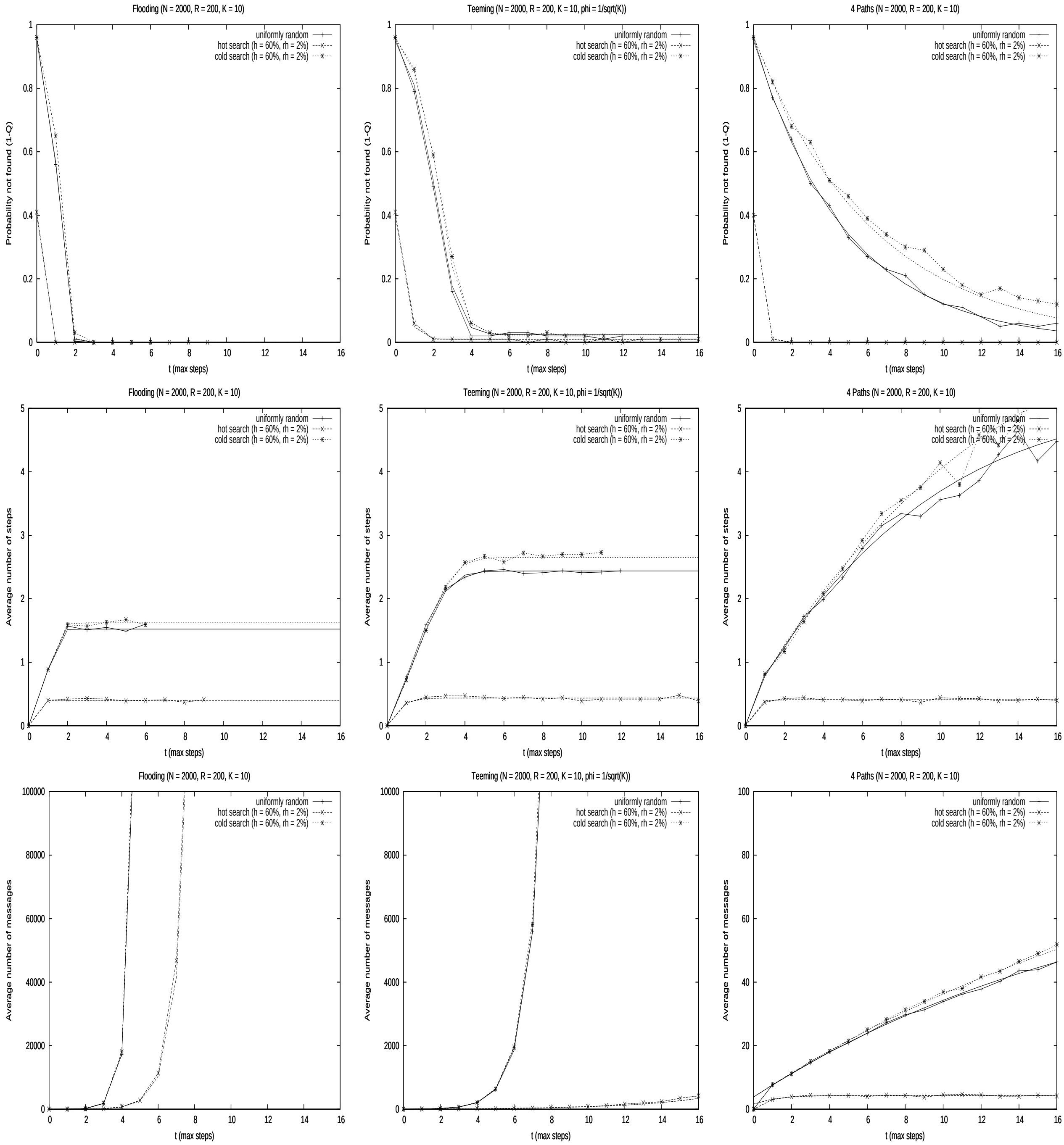

Figure 4. *Simulation and theoretical results for the proposed algorithms (I)*

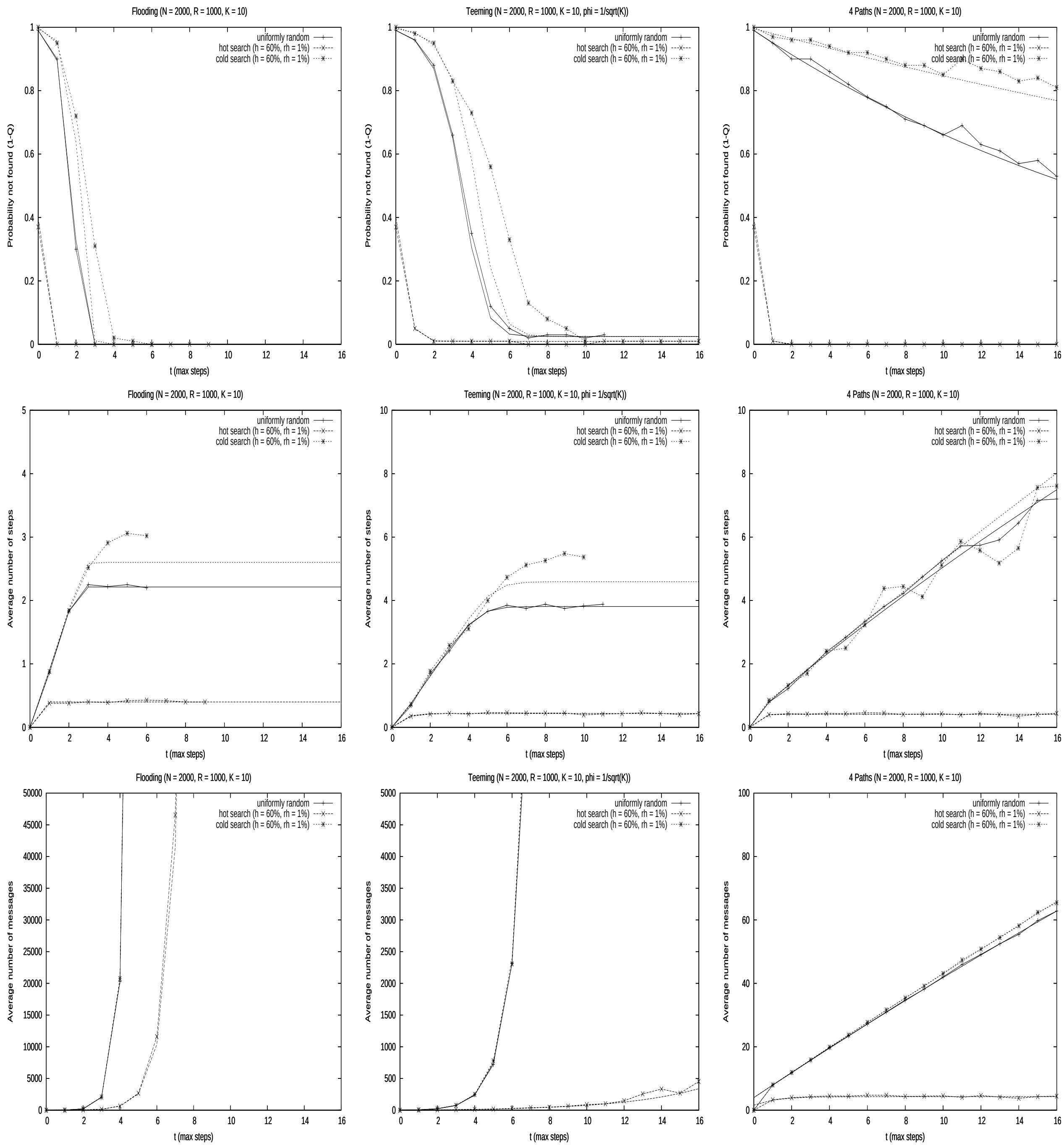

Figure 5. *Simulation and theoretical results for the proposed algorithms (II)*

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.

5.1. 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?

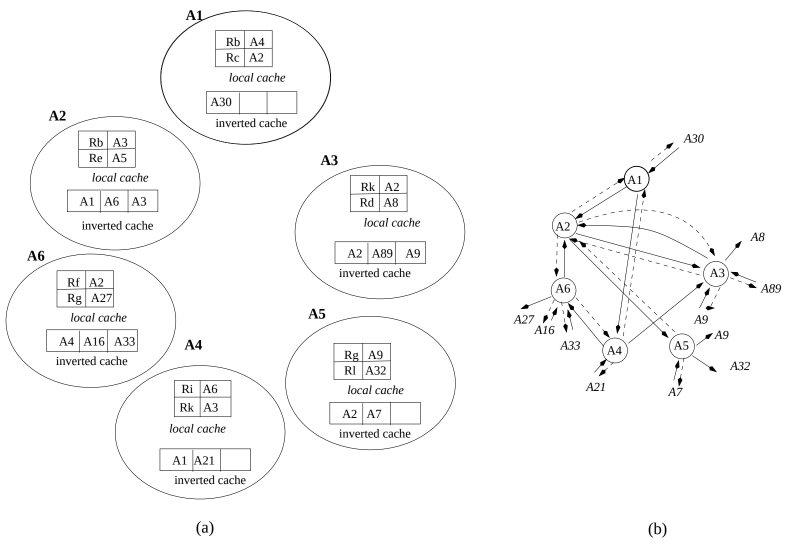

Figure 6. *The extended cache: (a) agents, (b) corresponding graph (dotted edges correspond to inverted cache entries)*

5.2. 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.

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.

6. Discussion

A topic that was not discussed in this paper is how the caches are initially populated; that is, how initially a node (agent) that enters the system learns about other resources.

An issue that we plan to study is how to make the algorithms adaptive; in particular how to adapt the number of nodes contacted so that the probability to locate a given resource increases. One approach is making the number of nodes that are contacted at a step i depend on i. In such a schema, the number of nodes contacted will increase or decrease with each step of the algorithm. Another approach is making the number of nodes depend on the number of resources, on the size of the cache, or on their ratio (a).

Another issue that we would like to explore is that of connectivity. In the presented work, we assumed that the cache network may be disconnected. An interesting question is: what are the conditions or protocols that must be enforced so that the network stays connected.

Yet another variation of the proposed algorithms could result by lifting the assumption that the network is directed. To achieve this, we require that when an agent (node) v knows an agent u, it contacts u so that u learns about v (u enters in its cache information about v).

In terms of comparing the performance of the proposed algorithms, we would like to see how our performance metrics (i.e., the probability of locating the resource and the average number of messages and steps to do so) vary with the number of paths (for the random path algorithms) and with ϕ (for teeming).

There are also a number of assumptions about the system model that we plan to investigate further. A rather straightforward extension is to assume variable cache sizes at the different nodes.

Finally, optimizations of the proposed algorithms may be devised. One way to reduce the number of messages for the flooding and teeming schemes is to eliminate the negative replies sent back to the inquiring node (this, as we already discussed, may require the use of timeout functions). If no negative replies are sent, then the only thing that changes in our analysis is the boundary condition of (9). A reply is sent back only if the resource is located in the node's cache, thus $m(0) = PC(1) = 1 - a$. It is not difficult to see, however, that the resulting number of message transmissions can be, in the best case, half of what (10) states. Thus, we may expect (at most) a 50% reduction in message transmissions but the rate remains exponential.

References

[1] M.A. Bauer and T. Wang. Strategies for Distributed Search. In *1992 ACM Conference on Communications, Kansas City, Missuri*, 1992.

[2] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource Discovery in Distributed Networks. In *PODC*, pages 229–337, 1999.

[3] Onn Shehory. A Scalable Agent Location Mechanism. In *Intelligent Agents VI, Agent Theories, Architectures, and Languages (ATAL), 6th International Workshop, ATAL '99, Orlando, Florida, USA, July 15-17, 1999, Proceedings*, volume 1757 of *Lecture Notes in Computer Science*. Springer, 2000.

[4] 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.

Appendix A. Flooding formulas

Eq. (7) Let $b = a^{1/(k-1)}$. Then:

$$
\begin{aligned}
Q_t^{(F)} &= \sum_{i=0}^{t} s_i^{(F)} \\
&= \sum_{i=0}^{t} \left(1 - a^{k^i}\right) a^{\frac{k^i - 1}{k-1}} \\
&= b^{-1} \sum_{i=0}^{t} \left(1 - a^{k^i}\right) a^{\frac{k^i}{k-1}} \\
&= b^{-1} \left(\sum_{i=0}^{t} a^{\frac{k^i}{k-1}} - \sum_{i=0}^{t} a^{k^i + \frac{k^i}{k-1}} \right)
\end{aligned}
$$

$$\begin{aligned}
&= b^{-1}\left(\sum_{i=0}^{t} b^{k^i} - \sum_{i=0}^{t} b^{k^{i+1}}\right) \\
&= b^{-1}\left(\sum_{i=0}^{t} b^{k^i} - \sum_{i=1}^{t+1} b^{k^i}\right) \\
&= b^{-1}\left(b^{k^0} - b^{k^{t+1}}\right) \\
&= 1 - b^{k^{t+1}-1} \\
&= 1 - a^{\frac{k^{t+1}-1}{k-1}}
\end{aligned}$$

Eq. (8) Letting $b = a^{1/(k-1)}$, and working exactly as above, we obtain:

$$\begin{aligned}
\overline{S_t^{(F)}} &= \frac{1}{Q_t}\sum_{i=1}^{t} i\left(1 - a^{k^i}\right) a^{\frac{k^i-1}{k-1}} \\
&= \frac{b^{-1}}{Q_t}\left(\sum_{i=1}^{t} ib^{k^i} - \sum_{i=1}^{t} ib^{k^{i+1}}\right) \\
&= \frac{b^{-1}}{Q_t}\left(\sum_{i=1}^{t} ib^{k^i} - \sum_{i=2}^{t+1}(i-1)b^{k^i}\right) \\
&= \frac{b^{-1}}{Q_t}\left(b^k - (t+1)b^{k^{t+1}} + \sum_{i=2}^{t+1} b^{k^i}\right).
\end{aligned}$$

Eq. (8) follows easily.

Appendix B. Teeming formulas

Eq. (12) We drop (T) from our notation for clarity. Using (3), we obtain:

$$\begin{aligned}
\overline{S_t} &= \frac{1}{Q_t}\sum_{i=1}^{t} is_i \\
&= \frac{1}{Q_t}\left(\sum_{i=1}^{t-1} is_i + ts_t\right) \\
&= \frac{1}{Q_t}(Q_{t-1}\overline{S_{t-1}} + ts_t).
\end{aligned}$$

From (2) it is seen that $Q_t = Q_{t-1} + s_t$, or, $s_t = Q_t - Q_{t-1}$. We thus obtain the following recursion on the number of steps:

$$\overline{S_t} = \frac{1}{Q_t}(Q_{t-1}\overline{S_{t-1}} + tQ_t - tQ_{t-1}$$

$$= \frac{Q_{t-1}}{Q_t}\overline{S_{t-1}} + t - t\frac{Q_{t-1}}{Q_t}.$$

Multiplying both sides by Q_t, we obtain:

$$Q_t\overline{S_t} = Q_{t-1}\overline{S_{t-1}} + tQ_t - tQ_{t-1}).$$

Letting $g_t = Q_t\overline{S_t}$, the recursion takes the following form:

$$g_t = g_{t-1} + tQ_t - tQ_{t-1},$$

with a boundary of $g_0 = S_0Q_0 = 0$. Easily,

$$\begin{aligned} g_t &= \sum_{i=1}^{t} iQ_i - \sum_{i=1}^{t} iQ_{i-1} \\ &= tQ_t - \sum_{i=0}^{t-1} Q_i, \end{aligned}$$

and, since $g_t = Q_t\overline{S_t}$, (12) follows.

Appendix C. Random paths formulas

Eq. (15) Using (14),

$$\begin{aligned} Q_t^{(p)} &= \sum_{i=0}^{t} s_i^{(p)} \\ &= 1 - a + a(1-a^p)\sum_{i=1}^{t}(a^p)^{i-1} \\ &= 1 - a + a(1-a^p)\sum_{i=0}^{t-1}(a^p)^{i} \\ &= 1 - a + a(1-a^p)\frac{1-a^{pt}}{1-a^p} \\ &= 1 - a^{pt+1}. \end{aligned}$$

where we $s_i^{(p)}$ is given by (14).

Eq. (16) Using (3) and (14),

$$\begin{aligned}
\overline{S_t^{(p)}} &= \frac{1}{Q_t^{(p)}} \sum_{i=1}^{t} ia(1-a^p)a^{p(i-1)} \\
&= \frac{a(1-a^p)}{Q_t^{(p)}} \sum_{i=1}^{t} i\,(a^p)^{i-1} \\
&= \frac{a(1-a^p)}{Q_t^{(p)}} \frac{-(t+1)(a^p)^t(1-a^p)+(1-(a^p)^{t+1})}{(1-a^p)^2} \\
&= \frac{-(t+1)a^{pt+1}(1-a^p)+a(1-a^{p(t+1)})}{(1-a^p)Q_t^{(p)}},
\end{aligned}$$

which, using (15) and after some straightforward manipulation gives (16).