Cache Updates in a Peer-to-Peer Network of Mobile Agents

Elias Leontiadis, Vassilios V. Dimakopoulos, Evaggelia Pitoura
Department of Computer Science, University of Ioannina
Ioannina, Greece GR-45110
{*ilias,dimako,pitoura*}*@cs.uoi.gr*

Abstract

In open multi-agent systems, agents need resources provided by other agents but they are not aware of which agents provide particular resources. We consider a peer-to-peer approach, in which each agent maintains a local cache with information about k resources, that is for each of the k resources, an agent that provides it. However, when an agent or a resource moves, cache entries become obsolete. We propose a suite of cache update policies that combine pull-based invalidation that is initiated by the agent that maintains the cache with push-based invalidation that is initiated by the agent that moves. We study and compare variations of oblivious flooding-based push/pull along with an informed push approach where each agent maintains a list of the agents that have it cached. Our experimental results indicate that a novel variation of flooding for push where a moving agent propagates its new location to agents in its old neighborhood achieves good cache consistency with a small message overhead. The proposed policies are suitable for any peer-to-peer system where peers cache information about other peers and this information becomes obsolete.

1. Introduction

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. Most solutions to this problem are based on a central directory that maintains a mapping between agents and resources [14]. However, such solutions do not scale well since the central directory becomes a bottleneck in terms of both performance and reliability.

In [4, 5], we proposed a peer-to-peer based 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 uses some form of flooding: it checks 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 information for the resource is found in some cache.

In this paper, we consider the case in which the agents are mobile and their contact information changes. This results in caches having incorrect information about the agents. We propose a suite of policies for addressing cache updates in this context. Note that to reach an agent that has moved, there must exist at least one cache in the network having the correct information about the agent's new location.

Our update policies combine pull-based invalidation that is initiated by the agent that maintains the cache with push-based invalidation that is initiated by the agent that moves. We consider periodic pull where an agent periodically updates its cache and on-demand pull where an agent updates a cache entry only when the entry becomes obsolete. With push, an agent informs other agents about its new location. We study variations of flooding-based push where an agent A that has moved propagates its new contact point to its neighbors blindly, that is, even if its neighbors do not have A in their cache.

We propose a novel variation of flooding, where agents that receive information about other moving agents maintains it for a short period of time in a *snooping* directory. Thus, with this method, a moving agent "infects" its old neighbors with its new address. We compare such oblivious flooding-based push approaches with an informed push where, in addition to its local cache, each agent A maintains a list, called inverted cache, of the agents that have A as their neighbor. When A moves, it pushes its new address only to agents in its inverted cache.

Our experimental results indicate that by fine-tuning the related parameters, snooping directories may lead to attaining the same cache consistency with plain flooding-based

push but with a ten-fold reduction of the message overhead. The inverted cache method is message-cost effective as compared to the oblivious push approaches, but only when cache replacements are infrequent.

The proposed policies are also applicable to a more generic context in which peers (agents in our case) cache information about the content of other peers. The problem we study here is the general problem of updating the caches when this information becomes obsolete, for instance when the peers are mobile as is the case with nodes in an ad-hoc wireless network.

The remainder of this paper is structured as follows. In Section 2, we present the peer-to-peer architecture of the cooperating agents as well as the search algorithms that form the basis for maintaining cache consistency. In Section 3, we introduce our cache consistency algorithms and in Section 4 we report our experimental results. In Section 5, we discuss related work. Finally, in Section 6 we offer conclusions and directions for future work.

2. System model

We assume a multi-agent system with nodes/agents, providing a number 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.

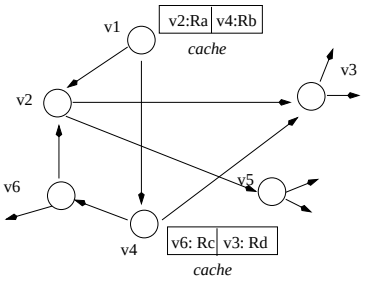

Figure 1. Part of a cache network, each agent v_i maintains contact information for $k = 2$ resources

Each agent is equipped with a private cache of size k. In there it stores information about k different resources, that is, for each of the k resources the contact information of one agent that provides it. 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 v to node u if and only if agent v has in its cache the contact information of agent u, in which case, u is called a neighbor of v. There is no knowledge about the size of V or E. An example is shown in Fig. 1.

In such a setting, a fundamental problem is that of resource location: how can an agent A, called the *inquiring agent*, 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 agents found in its cache, that is, some of A's neighbors, which in turn propagate the query to their neighbors and so on. For performance reasons, the search is usually limited to a maximum number of steps t, also known as time-to-live (TTL). A number of different approaches to this search procedure were proposed for example in [13, 4].

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, thus propagating the inquiring message. As the messages are sent from node to node, a "tree" is unfolded rooted at the inquiring agent (Fig. 2(a)). Note that the term "tree" is not accurate in graph-theoretic terms since a node may be contacted by two or more other nodes but helps to visualize the situation.

The most important disadvantage of flooding is the excessive number of messages that have to be transmitted, especially if t is not small. This number grows exponentially with the network density (cache size).

Teeming. To reduce the number of messages, a probabilistic variation of flooding called *teeming* can be applied. At each step of teeming, 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 (Fig. 2(b)). A node in the search tree may have from 0 to k children, $k\phi$ being the average case. Flooding is a special case of teeming for which $\phi = 1$.

Teeming with decay. Flooding and teeming suffer from a high percentage of duplicate messages, since intermediate nodes may receive the same query through different paths. To reduce this overhead, the fan out of the search in later steps should be decreased. Thus ϕ should not remain constant; it should rather decrease as the search progresses. In this policy, ϕ is a function $\phi = f(s, d)$ where s is the current step and d a decay parameter. We have experimented with various decay functions and the best results were obtained when ϕ decreases exponentially as the step increases: $\phi = (1 - d)^s$, where $d < 1$.

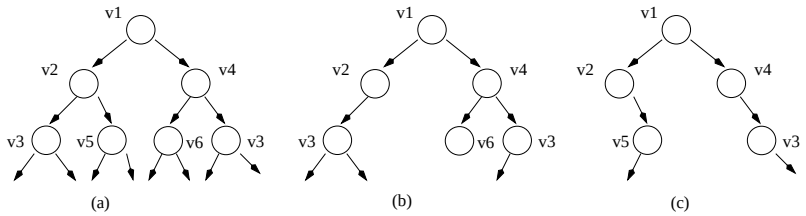

Figure 2. Searching the cache network of Figure 1: (a) flooding, (b) teeming, (c) random paths ($p = 2$)

Random paths. In this algorithm, the inquiring agent contacts $p \leq k$ of its neighbors. However, each intermediate node contacts only one of its own neighbors (randomly). The search space formed ends up being p random paths unfolding concurrently in the network (Fig. 2(c)). The algorithm produces less messages than flooding or teeming but it also needs more steps to locate a resource, in general.

3. Cache consistency algorithms

In mobile agent systems, agents or resources may change location at will. In such an environment, some agents will end up having invalid contact information about their cached resources. Invalid cache entries not only require renewed searches for the resource, but also deteriorate the performance of search algorithms. We propose a number of strategies to deal with the cache consistency problem. There are two basic categories of algorithms: push-based ones, which are initiated by the moving agents and pull-based ones which are initiated by the agents that have invalid cache entries. All proposed strategies use a combination of push and pull-based methods.

3.1. Pull

Pull-based updates are initiated by agents interested in refreshing their caches. Whenever an agent discovers that a cached resource has invalid contact information (i.e. the provider of the resource has moved), it initiates a search for the required resource. Any of the search algorithms described in the previous section can be utilized.

In general, given that at least one agent knows the correct location of the moving agent the agent that initiates the search should ultimately find the mover's new location. However, the search procedure may yield diverse results, for two reasons. First, by the time the correct location is communicated back to the inquiring agent, the mover may have moved again. Second, other agents may reply with stale contact information because their own caches have not been updated.

In any case, invalid results require the initiation of another search phase, which can lead to an excessive overhead. We can reduce this overhead by using *sequence numbers*. Each time an agent moves, it increases its own sequence number by 1 and communicates it along with its new location information. Pulling agents select the search replies with the largest sequence number per provider.

In addition to the above on-demand pull, agents can periodically pull their entire cache from the network. This is a prefetch operation aiming at avoiding the related search latency when the agent will actually be in the need of a resource. As a side effect, because periodic pulls tend to maintain cache entries up-to-date most of the time, on-demand pulls will also yield faster and more precise answers. However, periodic pulls cannot be too frequent, because too many pulls could cause high network traffic. Agents should pull their cache when they realize that a large portion is invalid.

3.2. Push

When an agent moves, it must inform at least one other agent about its new location. This is to say that pull alone cannot solve the consistency problem. The general idea is that the moving agent pushes (broadcasts) a message containing its new location to the network in order to inform as many agents as possible[1]. Next, we present and compare different strategies for implementing a push, including oblivious and informed ones, depending on the moving agent's knowledge about the network.

Plain push

In plain push, the agent broadcasts a message containing its new location. That is, it sends the message to its neighbors, that propagate it to their neighbors and so on. Push can be implemented using any of the algorithms presented in Section 2.

This strategy is oblivious since the moving agent is ignorant of which agent needs its new location; the idea is that by flooding the network, most of the agents that have cached the mover will be notified. For the plain push to inform most of the interested agents with high probability, it will have to utilize the flooding or the teeming algorithm

[1] A sequence number is also included in the message in order to assist the pull phases as discussed in Section 3.1

with a relatively large number of steps (so as to cover a good percentage of the nodes in the network).

Interested agents that will not be reached by the push will have to enter a pull phase. There is a tradeoff here; if the push utilizes a large number of steps, the pull phase can use some lighter search algorithm such as the random paths or a very shallow teeming, and vice versa. In general, if agents do not move very frequently, a push with high message overhead is tolerable so as to allow for lighter pulls.

Push with snooping directories

Plain push can be improved if combined appropriately with periodic pulling. The proposed strategy requires that each agent maintains a directory of recently moved agents (termed *snooping directory*).

As with the plain push algorithm, moving agents push obliviously their new location to the network. The difference is that any agent receiving a push message stores it in its snooping directory for a small period of time (termed *expiration time*), even if the agent is not interested in the mover.

When a pull phase is entered, agents consult their snooping directory before initiating a search. However, the snooping directory may give unreliable information if the agent pulls after a long period of time. Thus, all agents should pull their entire cache periodically in order to obtain possible updates. The interval between pulls should be less or equal to the expiration time.

Agents do not have to pull information from the entire network but pull using a small TTL instead. At the same time, this strategy allows for more relaxed algorithms for push, too. For example one could use a smaller flooding depth or a stricter decay parameter as compared to plain push. By appropriately tuning the involved parameters (push and pull algorithms, directory size, expiration time, pulling frequency) push with snooping directories should be a significant improvement over plain push.

Inverted cache

A way to avoid the potentially high message overhead of the push phase is to have some knowledge of where to send the updates. The basic idea behind this informed push strategy is that each agent should keep a directory of the agents to which it is known, called *inverted cache*.

When the agent moves, it only needs to inform the agents in its inverted cache; this is trivially optimal and at the same time eliminates the pulls. However, storing a full directory may not be always preferable. For example, popular resources or agents (that are cached by many others), may need to maintain a prohibitively large inverted cache. Thus only a limited directory may be maintained. As a result, not every interested agent will be updated by the moving agent, which means that pull phases will also be needed.

The inverted cache strategy can be combined with leasing. Leasing was proposed in [6] to ensure that a download is valid for a certain (leased) period of time. In our case, the lease duration indicates the time interval during which the resource owner guarantees that it will notify the leaser in case the former moves. Once the lease time expires, a leaser has to renew the lease so as to be informed of a possible move; otherwise the resource owner has the choice of evicting the leaser from its inverted cache.

Since the inverted cache maintains agents with valid leases, shorter leases imply smaller space overhead on average. The value of the lease time should be based on the movement frequency and the popularity of the agent. If the agent is highly mobile or very popular, the lease time should be small to keep the size of the inverted cache directory small.

When a leaser removes or replaces a resource from its cache it must contact the resource owner so that the latter deletes the leaser from the inverted cache. In addition, when a leaser moves it must contact all its neighbors (resource owners it knows about) so that the latter update the corresponding contact info in their inverted caches. Thus, when an agent moves it must contact all neighbors in its cache in addition to the agents in its inverted cache.

The proposed algorithm is very efficient especially when agents move frequently, since the updates become known quickly and with minimum overhead. However, it has increased memory requirements and it should be avoided in systems were the agents frequently change their cache. In addition, resource owners are forced to store and maintain state information about the agents that need them.

4. Experimental Results

To evaluate the performance of the proposed cache consistency protocols, we simulated a peer-to-peer network of agent caches. Each agent owns a number of resources and has a fixed cache size of k other agents/resources. We start our simulation by constructing a random graph of known agents: each agent picks k of the resources randomly and caches their location. We designate some resources as popular and also determine the extent of this popularity. A resource is offered by one agent only but an agent may hold many resources. All agents start with valid caches. Our simulation runs for a number of rounds (turns). During each turn, an agent may move or add/replace/remove a resource from its cache with a given probability.

Frequent cache replacements result in fewer invalid cache entries, so we kept the cache replacement probability quite low. By doing so, at the end of the simulation runs, the percentage of consistent cache entries is affected mainly by the cache updates algorithms.

The reported experiments are for a network of 1000

agents owning 3000 resources for 250 turns. The popular resources are few (2%) but they are known to many other agents (ten times more agents on average). Table 1 summarizes our parameters.

During the simulation sessions, we keep statistics regarding the message overhead of the pull and push algorithms, the percentage of cache entries that are valid at each agent, the maximum and average directory size of the agents (inverted cache or snooping directory). Each experiment was repeated 10 times and the results were averaged.

Table 1. Default simulation values

Parameter	*Default Value*
Number of agents	1000
Number of resources	3000
Number of turns	250
Cache size k	8 resources
Percentage of popular agents	2%
Popularity factor	10x more popular
Resources per agent	1 to 8
Probability to move at each turn	0.1%
Probability to replace a cached entry	0.6%

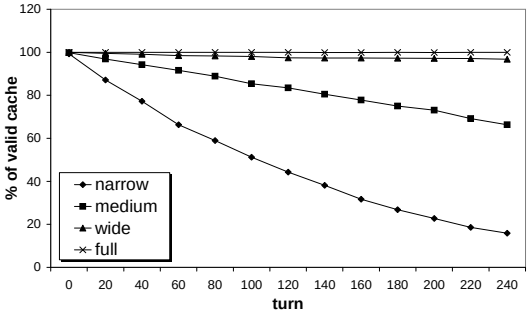

Figure 3. Narrow plain push with on-demand pull

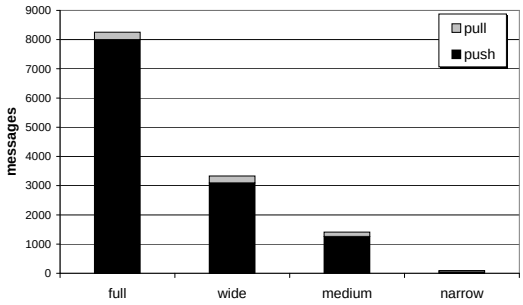

Figure 4. Plain push with on-demand pull: Average number of push and pull messages per move

4.1. Plain push

We experimented with a variety of algorithms for push (i.e., plain flooding, teeming, random paths). We report our experiments using teeming with decay, which exhibited better results. The main factor that influences the performance of plain push is the extent of flooding. We use the following decay function:

$$\phi = (1 - d)^s$$

By varying the decay parameter d and the maximum step t, we can control the extent of flooding. We used the values shown in Table 2. Agents pull information using a light k-random paths algorithms with $t = 3$.

Table 2. Push versions

Extent of push flooding	d	t
Full flooding	0.0	5
Wide flooding	0.2	5
Medium flooding	0.3	5
Narrow flooding	0.4	4

The moving frequency has only linear impact on the message overhead and the cache quality. The important issue in this method is the extent of flooding, since it determines the percentage of agents that receive an update. As shown in Fig. 3, a restrained push algorithm results in a poor cache condition at the end of the simulation. With wide flooding, we achieve satisfactory results: more than 98% of the cache is eventually consistent. Notice, that these results were obtained using very low cache replacement probabilities. With higher replacement probabilities cache validity would not tend to zero.

A wider push induces increased message overhead as shown in Fig. 4. The message overhead increases exponentially with the extent of push. The pull messages decrease too; this is because pull messages do not propagate further when a cache has invalid entries. That is why in the narrow push, where the majority of the cache entries is invalid, the pull messages are very few. Notice also that the cache consistency percentage of wide flooding is almost the same with full flooding, while, wide flooding has less than half the message overhead of full flooding. Thus fine-tuning the extent of push is important for getting balanced performance: good cache consistency with minimum message overhead.

The cache size k also affects the efficiency of the algorithm as depicted in Fig. 5. Narrow flooding performs poorly in sparse networks (small caches) and satisfactorily in dense networks (large caches). However, dense networks produce a large message traffic (exponential increase). So in dense networks, we may restrict the extent of push to re-

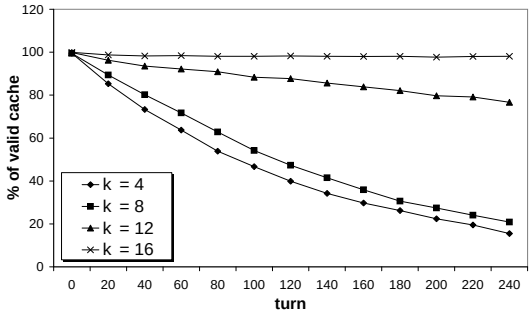

Figure 5. Plain push with on-demand pull: Cache size affects flooding performance

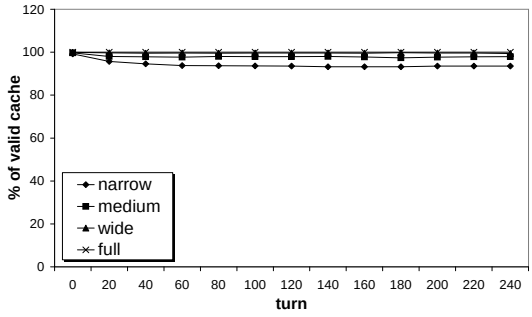

Figure 6. Snooping: Cache consistency for different extents of push flooding

duce the message overhead and still attain the same quality of cache consistency.

4.2. Snooping

With snooping, agents periodically pull information. In our simulation, the frequency of pull was set equal to the expiration time of the snooping directory, namely 20 turns. We used a very limited pull phase; we search only the 1-hop neighborhood for recent updates. So, we expect the pull message overhead to be small. We used the same push algorithm as in plain push since it gives the best results. As with plain push, the main parameter that affects the efficiency of snooping is the extent of push.

As shown in Fig. 6, with the same simulation parameters, snooping results in better cache consistency than plain push, while the message overhead is only slightly higher (Fig. 7). This is because, the pull phase frequently finds up-to-date information in the snooping directory. The pull message overhead is quite small. Thus, snooping allows us to use a more restricted push for achieving the same cache quality. In particular, as we can see in Figs. 3 and 6, we can use snooping with narrow flooding instead of a plain push with wide flooding to achieve more than 90% of valid caches. Snooping achieves this with ten times less messages than plain push.

4.3. Inverted cache with leasing

For these experiments, we used four values for the lease time: small (5 turns), medium (25 turns), large (50 turns) and very large (100 turns). We assume that agents do not renew their lease times. Expired entries are deleted from the inverted cache when its size becomes larger than $k/2$. Thus a minimum directory of $k/2$ agents is always kept. This is the only algorithm that cache replacements induce message overhead.

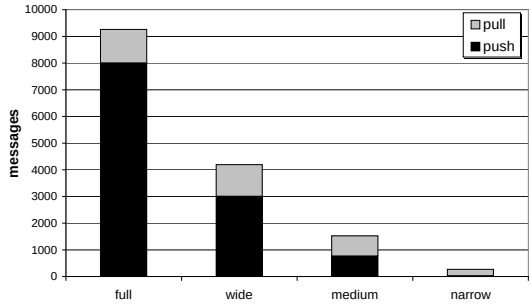

Figure 7. Snooping: Average number of push and pull messages per move

The leasing time affects the final consistency and the size of the inverted cache directory. Figure 8 shows how we can achieve a desired final consistency using appropriate leasing values. With inverted caches, an update is propagated with the minimum latency, since when an agent moves, it only informs the agents in its inverted and normal cache thus producing an 1-hop message overhead for push. However, message overhead is created by cache replacements, as shown in Fig. 9. Still, the messages increase linearly with the number of cache replacements. The inverted cache size is affected by the popularity of an agent. Popular agents have larger directories even if we use smaller leasing values for them. Additionally, the density of the network (cache size) affects the inverted cache size because dense networks result in more popular agents.

4.4 Discussion

When we do not want to use any additional memory for cache consistency, plain push is the only solution. Its performance is satisfactory when wide flooding is utilized. Since an agent that ends up with an invalid cache entry for a resource can pull on demand in order to locate the owner,

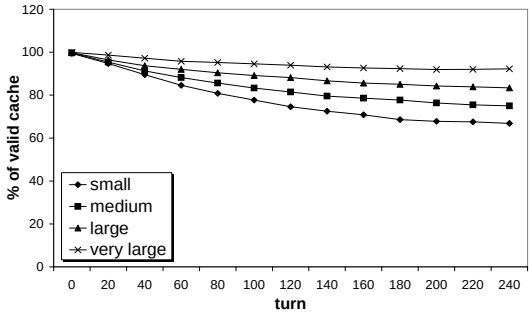

Figure 8. Inverted cache with leasing: Cache consistency for different leasing times

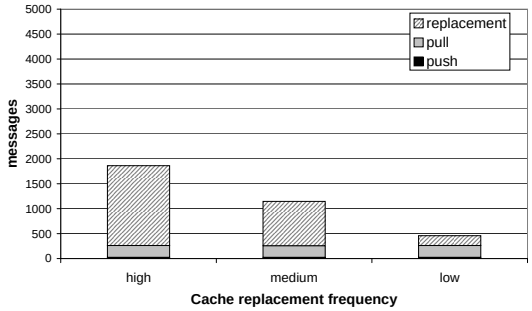

Figure 9. Inverted cache with leasing: Message overhead for different cache replacement frequencies

push can be used in dynamic peer-to-peer systems where agents (peers) go offline frequently. However, push has an increased message overhead when compared to the other schemes, so it should be avoided in the case of frequently moving agents.

The snooping method with periodic pulling is more efficient since it achieves the same cache quality with plain push but with a narrower flooding. As indicated by our simulation results, it may achieve the same cache consistency with plain push but with a message overhead an order of magnitude smaller. However, we must use additional memory (for the snooping directory) to track recently moved agents. This directory can become large in the case of highly mobile agents and dense networks.

The push phase of the inverted cache method induces a small overhead since only the neighbors in the cache and the inverted cache are contacted. Our experiments indicate that this overhead is almost a hundred times less than that of simple push. Moreover, cache updates are propagated immediately since we only use 1-hop communications. However, this is the only method for which cache replacements produce message overhead. Replacing a cache entry for a resource requires contacting the resource owner. As such, the inverted cache method is unsuitable for systems with a high replacement-to-mobility ratio. Our experiments demonstrate that the inverted cache method induces a smaller message overhead than the other two methods when this ratio is lower than 100 (that is, when we have at most 100 times more cache replacements than moves). The size of the inverted cache directory depends on the density of the network and the lease time and can become quite large for popular agents. This method is not appropriate for unreliable open MAS, because it requires agents to be online for maintaining a valid inverted cache and agents rely on each other to be updated. Finally, agents waste resources for keeping other agents informed.

5. Related work

The peer-to-peer cache network for agents was first proposed in [4]. However, [4] did not consider the fact that cache entries may become obsolete. In this paper, we focus on this problem, that is on maintaining consistency of the caches.

Distributed cache consistency has been the focus of much research [9, 15, 1, 16]. In general, most of the proposed solutions are based on a stateful server (i.e. a server that keeps information about the sites that maintain the cache entries) and thus apply some form of informed push. Most methods assume a reliable environment and cannot be applied directly to fully distributed and dynamic open MAS.

Our proposed methods are reminiscent of the epidemic algorithms first proposed for replicated databases [3]. Rumor spreading [10, 8] and gossip [12] methods are example epidemic algorithms used for lazy transmission of updates to distributed copies of a database. Both assume a peer-to-peer model for probabilistic update propagation among the sites that replicate files. In our case, this update may correspond to the new location of a resource. Push/pull algorithms were used in [2, 11] to make file updates known to all the peers in highly unreliable peer-to-peer systems like Gnutella [7].

Our algorithms differ in the following ways: (i) the update that we want to propagate is the new location of a resource/agent and not a file update or a database replica, and (ii) we use a combination of push and pull algorithms to achieve probabilistic cache consistency in the entire network. To this end, we extend the push algorithms proposed in [2] with two novel push methods (snooping push and inverted cache push) and combine them with pull to achieve efficient hybrid push/pull strategies.

6. Conclusion

In this paper, we consider the problem of cache updates in a peer-to-peer network of mobile agents. Each agent maintains in its cache information about other agents. When agents move, cached entries about them become obsolete. We propose a number of update policies that combine: (i) pull-based invalidation, that is initiated by the agent that maintains the cache with (ii) push-based invalidation that is initiated by the agent that moves. Both push and pull methods use appropriate variations of probabilistic message flooding. We propose a novel variation of push, where agents that receive information about other moving agents maintain it for a short period of time in a snooping directory. Our experimental results designate that by fine-tuning the related parameters, maintaining a snooping directory leads to attaining the same cache consistency with plain flooding-based push but with a ten times reduction of the message overhead. We also compare variations of oblivious push with an informed push approach where each agent maintains a list, called inverted cache, of the agents that have it cached. Our experiments indicate that the inverted cache method is message-cost effective when compared to the oblivious push approaches, but only when cache replacements are not frequent.

Our future plans include extending our approach for other types of mobile peers besides mobile agents. In particular, we are interested in peer-to-peer systems where peers are mobile nodes in an ad-hoc network. In this case, we plan to extend our cache update policies by taking into account information about how the location changes, e.g., the direction of a mobile peer's movement.

References

[1] P. Cao and C. Liu, "Maintaining strong cache consistency in the world wide web," *IEEE Transactions on Computers*, vol. 47, no. 4, pp. 445–457, Apr. 1998.

[2] A. Datta, M. Hauswirth, and K. Aberer, "Updates in highly unreliable, replicated peer-to-peer systems," in *Proc. of ICDCS 2003, 23rd International Conference on Distributed Computing Systems*, Providence, Rhode Island, May 2003, pp. 76–85.

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proc. of PODC 1987, 6th Annual ACM Symposium on Principles of Distributed Computing*, Vancouver, Canada, Aug. 1987, pp. 1–12.

[4] V. V. Dimakopoulos and E. Pitoura, "Performance analysis of distributed search in open agent system," in *Proc. IPDPS '03, International Parallel and Distributed Processing Symposium*, Nice, France, May 2003.

[5] ——, "A Peer-to-Peer Approach to Resource Discovery in Multi-Agent Systems," in *Proc. CIA 2003, 8th Int'l Workshop on Cooperative Information Agents*, Helsinki, Finland, Aug. 2003, pp. 62–77.

[6] C. Gary and D. Cheriton., "An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," in *Proc of SOSP 1989, the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park, Arizona, Dec. 1989, pp. 202–210.

[7] Gnutella Protocol Specification, Version 0.4, in *http://www.limewire.com/developer/*.

[8] R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek, "Rumor: Mobile data access through optimistic peer-to-peer replication," in *Advances in Database Technologies, ER '98 Workshops*, ser. LNCS, vol. 1552. Singapore: Springer, Nov. 1998, pp. 254–265.

[9] A. Kahol, S. Khurana, S. Gupta, and P. Srimani, "A strategy to manage cache consistency in a distributed mobile wireless environment," in *Proc. ICDCS 2000, Int'l Conf. Distributed Computing Systems*, Taipei,Taiwan, Apr. 2000, pp. 530–537.

[10] R. M. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking, "Randomized rumor spreading," in *Proc. FOCS 2000, 41st Annual Symposium on Foundations of Computer Science*, Redondo Beach, CA, Nov. 2000, pp. 565–574.

[11] J. Lan, X. Liu, P. Shenoy, and K. Ramamritham, "Consistency maintenance in peer-to-peer file sharing networks," in *Proc. of WIAPP'03, 3rd IEEE Workshop on Internet Applications*, San Jose, CA, June 2003, pp. 76–85.

[12] M.-J. Lin and K. Marzullo, "Directional gossip: Gossip in a wide area network," in *Proc. EDCC-3, 3rd European Dependable Computing Conference*, ser. Lecture Notes in Computer Science, vol. 1667. Springer, Sept. 1999, pp. 364–379.

[13] D. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, "Search and Replication in Unstructured Peer-to-Peer Networks," in *Proceeding of the 16th ACM International Conference on Supercomputing*, 2002.

[14] K. Sycara, M. Klusch, S. Widoff, and J. Lu, "Dynamic Service Matchmaking Among Agents in Open Information Environments," *SIGMOD Record*, vol. 28, no. 1, pp. 47–53, March 1999.

[15] J. Xu, X. Tang, D. L. Lee, and Q. Hu, "Cache coherency in location-dependent information services for mobile environment," in *Proc. MDA'99, First International Conference on Mobile Data Access*, ser. LNCS, vol. 1748. Hong Kong, China: Springer, Dec. 1999, pp. 182–193.

[16] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar, "Engineering server-driven consistency for large-scale dynamic web services," in *Proc. WWW 10, the 10th Int'l World Wide Web Conference*, Hong Kong, China, May 2001, pp. 45–57.