Creating and maintaining replicas in unstructured peer-to-peer systems*

Elias Leontiadis[1], Vassilios V. Dimakopoulos[2], and Evaggelia Pitoura[2]

[1] Department of Computer Science, University College London, United Kingdom
[2] Department of Computer Science, University of Ioannina, Ioannina, Greece

Abstract. In peer-to-peer systems, replication is an important issue as it improves search performance and data availability. It has been shown that optimal replication is attained when the number of replicas per item is proportional to the square root of their popularity. In this paper, we focus on updates in the case of optimal replication. In particular, we propose a new practical strategy for achieving square root replication called pull-then-push replication (PtP). With PtP, after a successful search, the requesting node enters a replicate-push phase where it transmits copies of the item to its neighbors. We show that updating the replicas can be significantly improved through an update-push phase where the node that created the copies propagates any updates it has received using similar parameters as in replicate-push. Our experimental results show that replicate-push coupled with an update-push strategy achieves good replica placement and consistency with small message overhead.

1 Introduction

The popularity of file sharing systems (such as Napster and Gnutella) has resulted in attracting much current research in peer-to-peer (p2p) systems. Peer-to-peer systems offer a means for sharing data among a large, diverse and dynamic population of users. An issue central in such systems is resource location, i.e. given a user query for data, to discover the peers with matching data items.

There are two basic approaches for building p2p systems for efficiently locating data. In structured p2p systems, data items are assigned to specific peers using some form of distributed hashing. Locating peers with matching data is then guaranteed to take place by visiting a bounded number of peers, normally logarithmic to the total number of peers in the system. In unstructured p2p systems, there is no assumption about the placement of data items. New nodes connect to some other nodes in the p2p system randomly. When compared with structured p2p systems, unstructured p2p systems usually provide no guarantees for search performance but do not suffer from the cost induced from maintaining the structure and from load balancing procedures necessary in structured p2p systems.

In this paper, we focus on the problem of replication in unstructured p2p systems. Replication improves the performance of search as well as data availability. Availability issues are especially critical in p2p systems, since peers leave the system very often,

* Work partially supported by the Integrated Project IST-15964 AEOLUS.

thus making their data unavailable. Previous work on the topic [1, 2] showed that optimal (with respect to search performance) replication is achieved when the number of copies per data item is proportional to the square root of their popularity. Here, we propose a new practical strategy for achieving square root replication called pull-then-push replication (PtP). With PtP replication, after a successful search for a data item, the node that posed the query enters a replicate-push phase during which it pushes copies of the item to its neighbors.

We also propose consistency maintenance protocols for copies created using the optimal replication strategy. We show that updating the copies can be significantly improved through an update-push phase where the node that created the copies propagates any updates it receives to its neighbors. Although, replica consistency protocols have been previously proposed (e.g., in [3]), our main contribution is that we study the problem in conjunction with the strategy used to create the copies. Our experiments show that the best results are achieved when update-push uses similar parameters with replicate-push.

2 Optimal replication

Suppose there are in total m different data items in the network, and that, collectively, the peers have capacity for storing R items[3]. Also, assume that the query rate for item i is q_i, $i = 1, \ldots, m$. Cohen and Shenker [1] developed a theory for optimally replicating the data items in unstructured peer-to-peer networks, given the restriction of R. In particular, they studied different replication strategies and showed that the expected search cost is minimized when the ith item has r_i replicas, where r_i is proportional to $\sqrt{q_i}$.

In their analysis the authors assumed a theoretical random probes (RP) search method: the inquiring node repeatedly probes peers in random and asks for the item, until the item is found. As the authors argued, the RP method captures the essential behavior of the blind search strategies (such as flooding) usually employed in p2p systems because in unstructured networks the topology is unrelated to the location of data. The problem with square-root (SR) replication is that it requires knowledge of the query rate for each item. To alleviate this, the following scheme was proposed: after each successful search, the item is copied to a number of nodes equal to the number of probes. It was shown that, with an analogous rate of item removals, this scheme leads to SR replication.

However, even this scheme is not easily implementable. Keeping track of the number of queried nodes is simply impractical when the usual flooding-based search algorithms are used, due to the excessive number of messages required. But even if a practical way of counting the queried nodes existed, this number would not be equal to the number of random probes that would have been required. The reason is that the theoretic RP strategy stops immediately after locating the item. All practical strategies, however, unleash parallel search paths — if the item is found in one of the search paths, the rest might continue querying nodes until, for example a time-to-live (TTL) parameter was exhausted.

[3] Data items can be actual copies of the data or just pointers to them.

In conclusion, practical strategies for approximating the number of probes are required. In [2], the authors examined a number of such algorithms, namely *owner-replication*, *path-replication* and *random-replication*. In owner-replication, the inquiring node is the only one that makes a copy of the resource — leading clearly to suboptimal replication. In the other two strategies, the node that provides the resource creates a number of replicas, equal to the distance (in hops) between the inquiring and the offering node. The last two strategies differ only in where the replicas are placed. Path and random replication approach SR replication but not quite accurately. The reason is that if the distance between the inquiring and the offering node is t hops, the RP strategy may not have located the item within just t probes, unless a single path was used for the search. The authors used multiple random walkers, which naturally visit a multiple of t nodes. We next propose a simple but effective scheme.

2.1 Pull-Then-Push replication

The proposed scheme is based on the following idea: the creation of replicas is delegated to the *inquiring* node, not the providing node. The scheme consists of two phases. The *pull* phase refers to searching for a data item. After a successful search, the inquiring node enters a *push* phase, whereby it transmits the data item to other nodes in the network in order to force creation of replicas. We call this the *Pull-then-Push* (PtP) replication. One can conceive variations of the PtP strategy by utilizing different algorithms for the pull and push phases. Path replication as suggested in [2] could be considered as a type of PtP replication, where the pull phase uses multiple random walkers, while the push phase uses a single path.

In order to reach SR replication, we need to create a number of replicas equal to the number of probed nodes. Consequently, one should utilize the *same* algorithm for the push and the pull phases, so that the push phase visits approximately the same nodes the pull phase visited. For example, if a random BFS search algorithm is used for the pull phase, the same algorithm should be used to broadcast the item during the push phase.

All practical search strategies produce multiple search routes, and utilize some form of TTL to limit the search space (and the resulting message overhead). If during the pull phase the item was found at distance t hops from the inquirer, then the push phase should also stop after t hops. This means that the TTL utilized for the push phases should not be set according to the TTL used during pull, but rather according to t.

However, because of the multiple search routes produced, the tth step may contact quite a large number of nodes. In [4], it was shown that for pure flooding, the number of messages grows exponentially with the TTL; most of those messages are sent in the last step of the search. For example, assume a random network with each peer connected to d other nodes, and a pure flooding strategy, where each peer propagates the query to all its neighbors. If a search returned an item at the 3rd step, approximately $d + d^2 + d^3$ different peers would have been visited, although only one node at distance 3 had the item. This means that $d+d^2+1$ probes could be enough and as a result, the best strategy for the push phase would be to use a TTL of 2, not 3. In general, the TTL used for the push phase should be equal to the hop distance at which the item was found minus one.

Recapping, our proposed PtP strategy adheres to the following rules: **(a)** After a successful search, the requester pushes the item back to the network; **(b)** The same

algorithm is used for both pull (search) and push; **(c)** The TTL for push is equal to $t-1$, where t is the hop distance where the resource was found; **(d)** All peers receiving the push message create a replica of the item. In the next section we provide simulation results which confirm that this simple PtP strategy does indeed lead to SR replication.

2.2 Experimental results

The PtP strategy has been evaluated through extensive simulations. In our simulator, we construct a network of peers/nodes, where each peer is connected to d other peers in random, called its neighbors. Each peer offers a number of data items and also has a fixed number of slots for replicating other items. Initially, all replica slots are empty. Then, we continuously perform searches originating at random peers, for random items. After each search, a push phase occurs, where replication is forced according to the strategy used. If a peer has to replicate an item and has no available slot, a uniformly random slot is emptied so that room is created for the new replica. Results are collected after a sufficiently large number of searches; the single most important metric we extract is the number of replicas, r_i, for each item.

The simulator is capable of utilizing a number of different search (pull) strategies. In all these strategies, a peer that receives a query for a data item, first checks whether it knows about the item; if not, it propagates the query to its neighbors. The strategies differ in the set of neighbors where the queries are propagated, and include [2, 4, 5]:

- *Pure flooding*. Peers propagate the query to all their neighbors.
- *Random walkers or random paths*. For a single random path, each peer propagates the query to exactly one of its neighbors, in random. Multiple walkers searching in parallel is a variation to decrease the average number of hops: the inquiring node sends the query to a number of its neighbors, each one unleashing a random walker.
- *Random BFS or teeming*. Peers propagate the query to each of their neighbors with some fixed probability ϕ. A *decay* parameter may be utilized so that ϕ decreases with the distance from the inquiring node. If a node is in distance t from the inquiring peer, then the probability of contacting a neighbor is given by: $\phi_t = \phi(1-c)^t$, where $\phi_0 = \phi$ and c is the decay parameter. For $c = 0$ we have simple teeming, while if in addition $\phi = 1$, the strategy is pure flooding.

The same algorithms are used for the push phase. Of course, in this case the peers do not receive queries but just items to propagate immediately to some of their neighbors.

In Fig. 1, we present results for a random network of 1000 peers, each with 4 neighbors on average. A peer has storage space for 10 items, out of a total of $R = 100$ different items. The replication strategies employed are owner, path and PtP replication. For PtP we experimented with all the algorithms presented above and with different parameters. In Fig. 1, we show the results for two of them, one with 5 random walkers and TTL = 10 and one with teeming, TTL = 5 and a decay parameter of $c = 0.4$. The other algorithms exhibited the same behavior, and were omitted for clarity. The plot shows the normalized number of replicas (r_i/R) for each of the items. To make the square-root trend clearer, for this particular plot, we have assumed query rates proportional to the

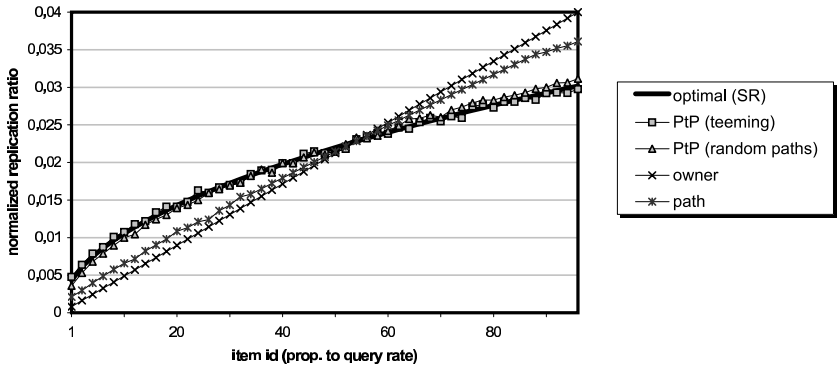

Fig. 1. Distribution of replication ratios under various replication strategies

id of the item, so the x-axis could also be named 'query rate'. The plot includes the optimal square-root distribution (SR), drawn with a thick line. We have also experimented with other query rates, including Zipf-like ones, and the results were identical.

It should be clear from Fig. 1, that owner replication is far from the optimum. Path replication is better, but does not result in SR replication. Both PtP strategies, although different by nature, led to almost perfect SR replication. This also comes to confirm our intuition that the exact strategies used for the pull/push phases of PtP are not very important, as long as they are the same in both phases. Here we only show PtP's ability to approximate SR replication. Results on PtP's performance and the achieved search gains can be found in [6].

3 Consistency maintenance

Replication induces the need for consistency maintenance, that is, keeping the replicas up to date whenever changes occur. For the discussion that follows, we assume that each data item has a single *owner*, which is also the single peer that is allowed to modify the item. Upon modification, the replicas which have been spread over the network must be made consistent with the most recent version of the data item.

The problem of consistency maintenance appears in many contexts [7, 8]. In [3, 9], various strategies were proposed in the context of peer-to-peer systems. In general, updates of a data item are broadcast by the owner and/or are searched for by the peers that have the replicas. Thus, solutions to the consistency maintenance problem utilize **(a)** owner-initiated update *push*, so that peers with replicas are communicated the update, **(b)** replica holder-initiated *pull*, either when needed or periodically, so as to discover new updates, if any, or **(c)** a combined push/pull scheme.

It has been shown that usually a combined push/pull strategy (P/P for short) constitutes the best tradeoff between consistency levels and message overhead [9, 5]. The owner performs a limited push of the updates and the peers pull periodically, just in case the owner-initiated push did not reach them.

A basic problem in these P/P protocols is when should a peer pull. Pulling too often creates substantial message overhead. Pulling infrequently may result in missing im-

portant updates. Adaptive pull strategies try to minimize the communication overhead, while maintaining good consistency levels by having each replica holder pull at specific intervals. These intervals are determined by a time-to-refresh (TTR) parameter, which is adaptively adjusted depending on the previous pull results. If after the last pull the item was found unchanged, TTR is increased so as to pull less frequently; otherwise, TTR is decreased so as to check for updates more often.

Our premise is that efficient consistency maintenance can be achieved only in conjunction with efficient replication. If the number of replicas and their placement is well-planned, then the algorithms for maintaining them under updates can be much more effective. To this end, we propose a novel push/pull update strategy that utilizes knowledge about replica creation so as to improve update efficiency. Our experiments have shown that consistency maintenance can be achieved quite efficiently when replication is done in the optimal way, using the PtP strategies. Optimal replication not only minimizes the average search costs but also reduces the average update costs when combined with a suitable update strategy.

3.1 Updates under optimal replication

From now on we assume that items have been replicated in the network and that replication has been done using the PtP strategy. As discussed earlier, the PtP strategy requires that, after a successful search, the peer that found the item creates a number of replicas, through a *replicate-push* phase, or R-push for short, with an appropriate TTL value. The basic idea now is to let this peer be held "responsible" for updating the replicas it created, as explained next. With respect to a particular data item, the nodes in the network fall into one of the following three categories:

- *owner*: the single peer that produces new versions of the data item
- *responsible*: a peer that searched for the item in the past (and thus forced the creation of replicas of the item)
- *indifferent*: a peer that was forced to hold a replica of the item.

The strategy, which we call PtPU, is a combination of push/pull. The owner broadcasts new updates to the network, through an update-push, or U-push for short. Whenever a "responsible" peer receives a new version of the item (either through an *update-pull* that it itself performed or an U-push that the item owner initiated), it undertakes the task of updating the replicas it created. In other words, it performs a U-push itself for the new version of the item. Moreover, this U-push should employ the same TTL parameter as the one used in the R-push, thereby reaching approximately the same nodes that were previously reached in order to create replicas.

This scheme has the potential of reducing the overhead of consistency maintenance significantly. A peer that is "responsible" for a resource should check (pull) frequently for newer updates of the item, using a smaller TTR value. Peers which were forced to have replicas of this item ("indifferent" peers) do not need to pull (or, they could pull quite infrequently; *cf* the discussion in Section 4), relying on some "responsible" peer to provide an update for them. Summarizing, our strategy behaves as follows:

- The owner pushes the new versions of the item

– "Responsible" peers pull periodically, and push any updates they become aware of to their neighborhood exactly as when they created the replicas (i.e. with the same parameters as in the push phase of PtP).
– The other peers do nothing; they rely on "responsible" peers to keep them updated.

For the periodic pulls of the "responsible" peers, we follow an adaptive scheme [9], whereby the time-to-pull-next (TTR) is decreased or increased according to the perceived version of the item. If the last pull did not return a newer version, the estimate for the next TTR will be increased by some constant: $\text{TTR}_e = \text{TTR} + C$. If, on the other hand, a more recent version of the item was found, the next TTR should be decreased. It should be decreased in proportion to the difference, D, in versions between the pulled item and the one the peer had — the higher the difference D, the more the missed updates, and hence the more frequent the pull should be. Thus, the estimate for the new TTR is: $\text{TTR}_e = \text{TTR}/(D + \beta)$, where β is a parameter that provides some reduction in TTR in the case of $D = 1$. The next TTR is a weighted average of the current TTR and the estimate:

$$\text{TTR} \longleftarrow w\text{TTR}_e + (1 - w)\text{TTR},$$

where, w is a parameter determining the rate of change — smaller values of w make TTR change very slowly, while larger values make TTR adapt quickly to variations.

3.2 Experimental Evaluation

We have evaluated the performance of both the P/P and the PtPU strategies through extensive simulations. The network of peers is constructed and the data items are replicated using the PtP strategy as described in Section 2.2. After creating the replicas, we initiate simulation sessions. Each session runs for a number of rounds (turns). During each turn, the owner of an item creates a new version of the item with a given update probability p_u (update rate) and pushes it to the network. In the P/P strategy, all peers with replicas pull for new versions using adaptive pull. With PtPU, only the "responsible" peers pull using, again, adaptive pull. In addition, the "responsible" peers push any received updates to their neighbors using exactly the same strategy used when the replicas were created (for example, using teeming with the same decay and TTL values).

We evaluate the performance of the update strategies with respect to two parameters: the achieved consistency and the associated message overhead. The consistency level is measured as the percentage of replicas that are up-to-date. We experimented with different strategies for propagating the updates (i.e., pure flooding, random walkers, teeming and teeming with decay). The results attained were qualitative the same, thus, we report here only the results obtained when using teeming with decay, which is the method that gives us the most flexibility in terms of tuning the extend of the propagation. In particular, we present results when using three variations of teeming as summarized in the table that follows. Wide teeming visits more peers, while narrow teeming produces smaller message overhead.

Extend of teeming	c (decay)	TTL
Wide	0.1	5
Medium	0.3	5
Narrow	0.4	4

Regarding the adaptive pull, the tuning of its parameters is beyond the scope of this paper. A set of values that were found to work well in adapting the TTR is: $w = 0.8$, $b = 0.5$, and $C = 10$ turns, and those are the values that were used in all the experiments presented here. The reader is referred to [10, 7] for a detailed discussion of the topic.

Performance with respect to the update rate. The goal of the first set of experiments is to depict the behavior of plain P/P and PtPU under different update rates. We consider two cases: frequent updates ($p_u = 0.1$), and infrequent updates ($p_u = 0.025$). The owner pushes the updates using narrow teeming. The reason for using such a rather limited push is to make the effect of pull more clear. To discover a general trend, we let both strategies utilize exactly the same pull characteristics (i.e. the same variations of teeming) and see how they compare with each other. The results are shown in Fig. 2 for high update rates and in Fig. 3 for infrequent updates. Each strategy is simulated for pulling with wide, medium and narrow teeming. In the case of high update rates, peers are forced to a high pull overhead in the P/P strategy so as to be frequently updated. In the PtPU case, though, pull is limited. Push messages are more since the "responsible" peers also propagate any updates they receive. For a low update rate, it is easier for any strategy to keep good consistency levels, utilizing fewer messages. Even in this case, though, PtPU achieved consistency levels above 92%, while plain P/P is, at best, a little above 80%. PtPU consistently outperforms P/P by any measure. It results in better consistency levels and, at the same time, fewer messages.

Comparison of the two update policies. In this set of experiments, we compare further the two methods. In particular, we show (i) the level of consistency achieved when the two methods produce the same number of messages and (ii) the number of messages required by each method for achieving the same consistency level. Here, we consider a medium update rate ($p_u = 0.05$). For each strategy we repeatedly alter the pull parameters until we achieve the same value for the metric of interest (i.e. the consistency level or the number of messages) among all strategies.

The results are presented in Figures 4–6. In the plots, we also consider the performance of P/P and PtPU, for the case where the creation of replicas does not follow the PtP strategy. Instead, after the replication phase, the replicas get scattered across the network. Our goal is to show that loosing the locality induced by the PtP strategy results in worsening the performance of both the P/P and PtPU strategies. Note that the number of replicas is kept the same; what differs is their placement in the network. The strategies under random placement of the replicas are marked with an "(R)" in the plots.

In Fig. 4 the owner uses a narrow push to propagate the updates. We run the simulator tuning the pull parameters until all strategies achieved approximately the same consistency level of 82%. The resulting message counts show that plain P/P required 43% more messages than PtPU to achieve the same consistency. In Fig. 5 the owner uses a medium push to propagate the updates, so as to make it easier for the inferior strategies to achieve higher consistency levels (but, of course, with higher message overhead). The achieved consistency levels where approximately 95%. Once again, plain P/P required 46% more messages than PtPU. In Fig. 6 all strategies generated approximately 62000 messages. PtPU required a narrow pull while P/P's adaptive pull resulted in a

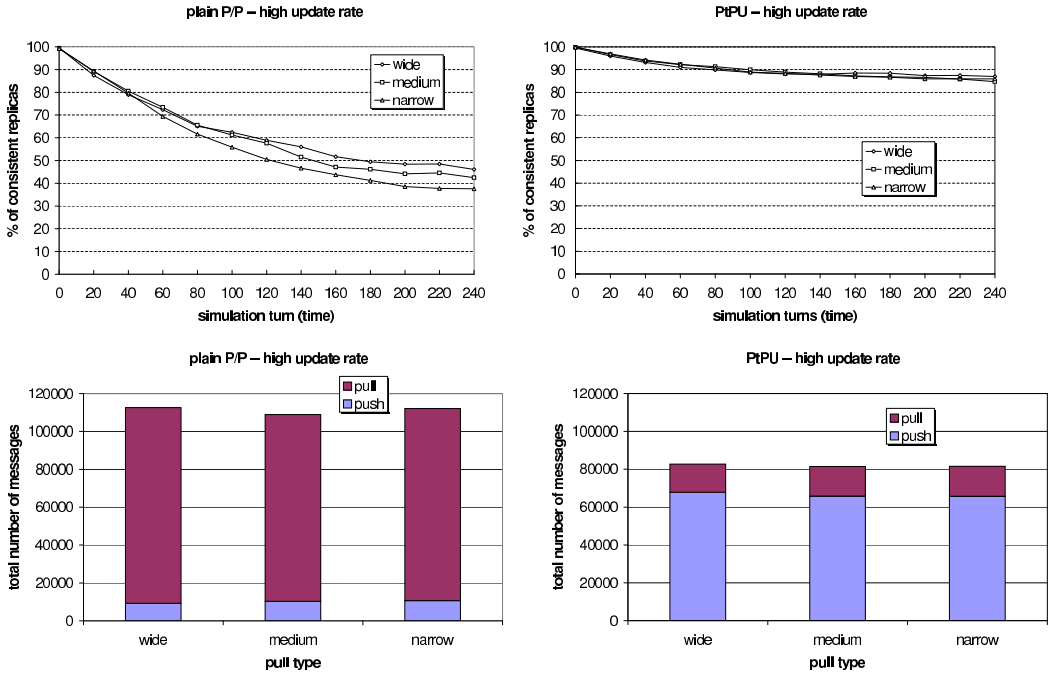

Fig. 2. Performance of the two strategies under high update rates.

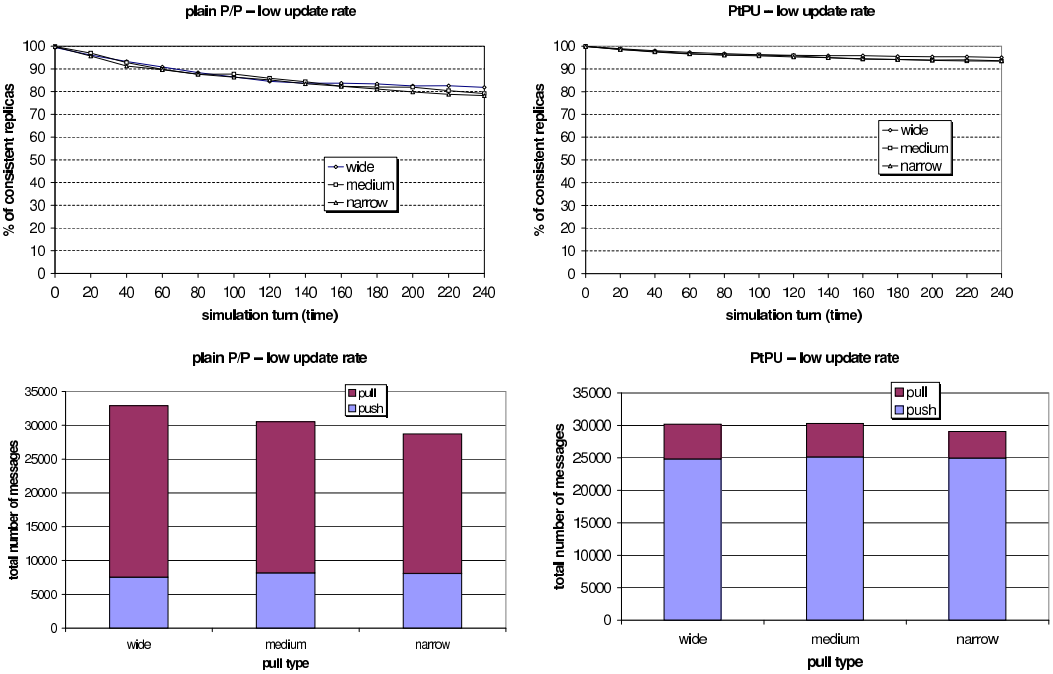

Fig. 3. Performance of the two strategies under low update rates.

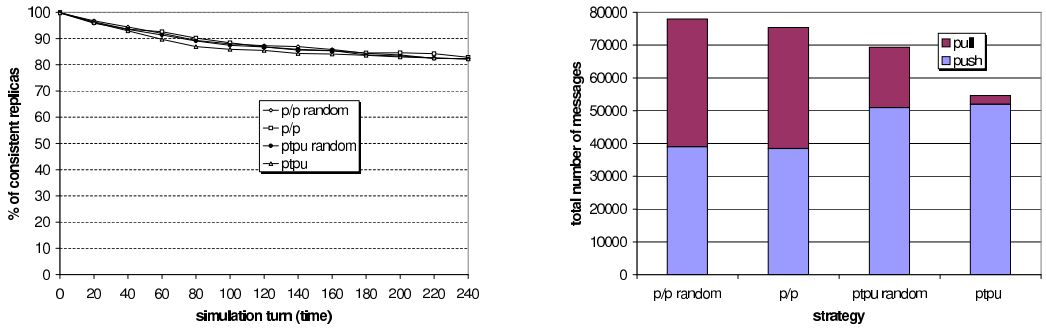

Fig. 4. Number of messages when all strategies result in consistency levels of approximately 82%.

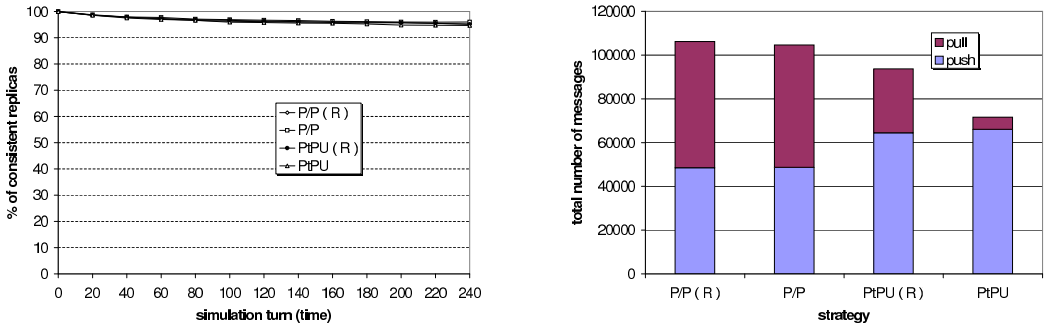

Fig. 5. Number of messages when all strategies result in consistency levels of approximately 95%.

wider teeming. The superiority of the PtPU strategy is shown vividly, as it managed to achieve more than 90% consistency.

Another conclusion from these plots is that, indeed, the random placement of replicas makes the performance of P/P and PtPU worse. This validates our intuition that the inherent locality of replica creation through PtP results in more efficient updates.

4 Discussion

In this paper, we consider replication in unstructured p2p systems. The idea behind our approach is that developing protocols for consistency maintenance which utilize knowledge about the strategy used to create the copies increases the efficiency of such protocols. Based on this, we develop a simple strategy for achieving square-root replication, which was previously proved to be optimal for unstructured peer-to-peer systems, and a consistency maintenance protocol that is tuned for our replication strategy.

Our experimental results show that our protocols achieve significantly better consistency for a smaller communication cost than protocols that do not exploit knowledge of the underlying replication strategy. A more detailed version of this work can be found in [6].

In our experiments we have assumed that the network does not change during the replication and update phases. We are currently studying the behavior of our strategies

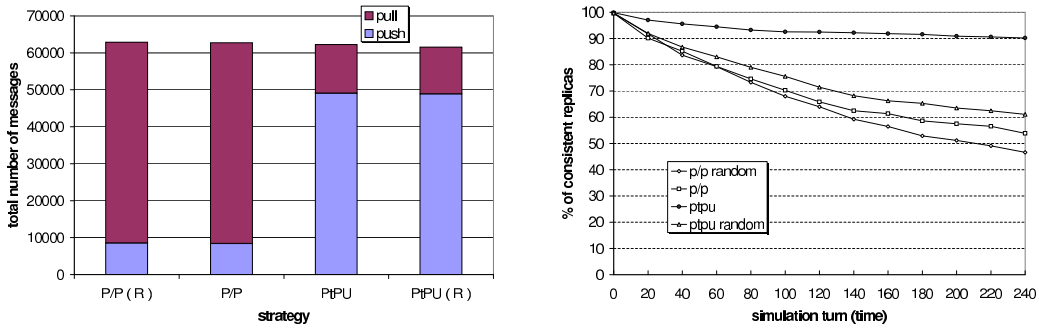

Fig. 6. Consistency quality when all strategies generate the same number of messages.

in more dynamic settings where peers enter or leave the system at will. In such environments the PtPU strategy may encounter the following problem: a "responsible" peer could depart from the network, leaving thus a number of "indifferent" nodes without anybody to update their replicas for them. Thus, it is almost imperative that "indifferent" peers should pull, too, just in case the "responsible" node is not near them anymore.

References

1. Cohen, E., Shenker, S.: Replication Strategies in Unstructured Peer-to-Peer Networks. In: Proc. ACM SIGCOMM'02. (2002)
2. Lv, D., Cao, P., Cohen, E., Li, K., Shenker, S.: Search and Replication in Unstructured Peer-to-Peer Networks. In: Proc. ICS'02, 16th ACM Int'l Conference on Supercomputing, New York, USA (2002)
3. Datta, A., Hauswirth, M., Aberer, K.: Updates in highly unreliable, replicated peer-to-peer systems. In: Proc. of ICDCS 2003, 23rd Int'l Conference on Distributed Computing Systems, Providence, Rhode Island (2003) 76–85
4. Dimakopoulos, V.V., Pitoura, E.: Performance analysis of distributed search in open agent system. In: Proc. IPDPS '03, Int'l Parallel and Distributed Processing Symposium, Nice, France (2003)
5. Leontiadis, E., Dimakopoulos, V.V., Pitoura, E.: Cache Updates in a Peer-to-Peer Network of Mobile Agents. In: Proc. P2P2005, 4th Int'l Conference on Peer to Peer Computing, Zurich, Switzerland (2004) 10–17
6. Leontiadis, E., Dimakopoulos, V.V., Pitoura, E.: Creating and Maintaining Replicas in Unstructured Peer-to-Peer Systems. Technical Report TR2006-01, Univ. of Ioannina, Dept. of Computer Science (2006)
7. Srinivasan, R., Liang, C., Ramamritham, K.: Maintaining temporal coherency of virtual data warehouses. In: Proc. RTSS '98, 19th Real Time Systems Symp., Madrid, Spain (1998)
8. Urgaonkar, B., Ninan, A., Raunak, M., Shenoy, R., Ramamritham, K.: Maintaining mutual consistency for cached web objects. In: Proc. ICDCS 2001, 21st Int'l Conference Distributed Computing Systems, Phoenix, AZ, USA (2001)
9. Lan, J., Liu, X., Shenoy, P., Ramamritham, K.: Consistency maintenance in peer-to-peer file sharing networks. In: Proc. of WIAPP'03, 3rd IEEE Workshop on Internet Applications, San Jose, CA, USA (2003) 76–85
10. Lan, J.: Cache Consistency Techniques for Peer-to-Peer File Sharing. Technical report, MSc Thesis, Dept. of Computer Science, Univ. of Massachusetts (2002)