# Near-Optimal Dynamic Replication in Unstructured Peer-to-Peer Networks

Mauro Sozio
Max-Planck-Institut für Informatik
Saarbrücken, Germany
msozio@mpi-inf.mpg.de

Thomas Neumann
Max-Planck-Institut für Informatik
Saarbrücken, Germany
neumann@mpi-inf.mpg.de

Gerhard Weikum
Max-Planck-Institut für Informatik
Saarbrücken, Germany
weikum@mpi-inf.mpg.de

## ABSTRACT

Replicating data in distributed systems is often needed for availability and performance. In unstructured peer-to-peer networks, with epidemic messaging for query routing, replicating popular data items is also crucial to ensure high probability of finding the data within a bounded search distance from the requestor. This paper considers such networks and aims to maximize the probability of successful search. Prior work along these lines has analyzed the optimal degrees of replication for data items with non-uniform but global request rates, but did not address the issue of where replicas should be placed and was very very limited in the capabilities for handling heterogeneity and dynamics of network and workload.

This paper presents the integrated P2R2 algorithm for dynamic replication that addresses all these issues, and determines both the degrees of replication and the placement of the replicas in a provably near-optimal way. We prove that the P2R2 algorithm can guarantee a successful-search probability that is within a factor of 2 of the optimal solution. The algorithm is efficient and can handle workload evolution. We prove that, whenever the access patterns are in steady state, our algorithm converges to the desired near-optimal placement. We further show by simulations that the convergence rate is fast and that our algorithm outperforms prior methods.

## Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

## General Terms

Algorithms, Theory

## 1. INTRODUCTION

### 1.1 Motivation

Replication is widely used in distributed data management with read-intensive workloads for a number of reasons.

First, replicating data items at different nodes in the network improves reliability (i.e., the probability of not losing any data even in the presence of multiple permanent node failures) and availability (i.e., the probability of having the data accessible even in the presence of multiple temporary node outages). Second, when access requests have a choice among multiple replicas, load balance can be improved and results in higher throughput and shorter response times. Third and most importantly for the scope of this paper, search requests in large-scale networks are often processed by partial search only, and replication then improves the *probability of successfully retrieving* the requested search result. A canonical example for this case are *unstructured peer-to-peer (P2P) overlay networks* such as Gnutella or BitTorrent with file sharing among large user communities. In these networks, queries are processed by various forms of epidemic messaging, most notably, bounded flooding where a query initiator sends requests to all or a randomly chosen subset of network neighbors and these requests are forwarded up to some prespecified depth or when a preset time-to-live (TTL) expires.

An alternative to unstructured P2P networks are structured overlays based on distributed hash tables (DHT's) [24]. These systems are also equipped with replication, and they can locate every item and its replicas with logarithmic space and time costs. Many of the issues that the current paper addresses in the context of unstructured networks apply to structured overlays as well; some issues are simpler with DHT's, others are more expensive. This paper focuses on unstructured P2P networks because practically adopted solutions are largely based on this paradigm (or use hybrid architectures). It seems that, although epidemic messaging is expensive in the worst case, its "typical-case" behavior is much better and very competitive (partly for the very reason that replicas are available).

The protocols for P2P replication management and processing search requests are fairly mature [21]. What is much less understood are strategies for the creation and placement of replicas in such large-scale, heterogeneous, highly dynamic networks. Two key questions are: how many replicas should a data item have, and where in the network do we place these. Good - ideally near-optimal - answers to these questions need to consider the request rate (popularity) of each data item, the distribution of query originators for the requests to an item, the network topology and per-link performance characteristics, the storage capacity of each peer, and many more parameters. Not surprisingly, the solutions proposed in the literature are based on much simpler computational models that reflect only a subset of these characteristics of realistic systems. One of the most elegant and widely recognized methods is the *square-root-replication* result by Cohen and Shenker [6]: for a given network and a set of data items, an optimal solution for minimizing the expected search length of successful requests is obtained by choosing the degrees of replication to be proportional to the square root of the global request rates of the data items. The question of where in the network the replicas should be placed is not addressed in this work. The implicit approach of a uniformly random choice would be adequate only if network links have uniform latency and bandwidth and the request rates to different items are the same for each peer.

Notwithstanding its depth and elegance, the square-root-replication result is still far from being practically applicable, and similar limitations hold for most of the literature (we will discuss related work in Section 2). In addition to not paying sufficient attention to system and workload heterogeneity, the dynamics of large P2P networks is a major issue that is typically considered only as an afterthought (for exceptions see Section 2). Not only can the workload evolve over time and then call for incremental re-optimization on the fly, but the P2P network itself usually exhibits high dynamics in the sense that peers can join and leave at any time without prior notice and possibly at a high rate. This so-called churn phenomenon requires dynamic adjustments to both replication degrees and the placement of the replicas. This paper addresses this issue and also overcomes several other limitations of the prior work on P2P replication models and algorithms.

### 1.2 Problem Definition

A P2P network is represented as an undirected graph $G = (V, E)$ with $n$ vertices; two vertices are neighbors in such a graph if and only if the corresponding two peers can directly communicate with each other. There are $m$ distinct data items in the system, each item $j$ having size $s_j$. Each peer, is willing to dedicate a portion of its storage capacity to improve network performance; we denote their capacity as $c_i$. The remaining capacity is used by each peer to store its own files which are referred to in the following as *untouchable* files, meaning that such files cannot be deleted by the replication algorithm.

For each peer $i$, we define $q_{ij}$ to be the fraction of all queries (issued in the whole network) for item $j$ by peer $i$. Hence, all query rates are non-negative; moreover: $\sum_{i=1}^{n}\sum_{j=1}^{m} q_{ij} = 1$.

A walk of length $k$ in the given graph, is a sequence of vertices $v_1, v_2 \ldots, v_k$ (not necessarily distinct), such that $(v_i, v_{i+1}) \in E\,(1 \leq i \leq k-1)$. A random walk of length $k$, starting in $v_1$, is a walk where at each step the vertex $v_i$ is chosen independently at random among the neighbors of $v_{i-1}$.

A query is defined formally as a pair $\langle W, j\rangle$ where $W$ is the random walk by which the query if forwarded to a peer (that would ideally hold a replica of item $j$) and $j$ is the item requested. We allow many queries for the same item delivered through the same random walk; each of them is associated with a unique *id*. In the following, a query for item $j$ is said to be successful, if the corresponding generated random walk traverses at least one peer that has a replica of item $j$.

Given an assignment of data replicas to peers, we model a search in the P2P network by the following stochastic process: at each step a query for item $j$ is issued by a peer $i$ with probability $q_{ij}$; This query generates a random walk of length at most $k$, which starts in $i$ and which ends as soon as the random walk comes across a peer containing item $j$.

We denote with $p_{ij}$ the probability that a peer $i$ is traversed by an unsuccessful random walk for item $j$ in this stochastic process; this is the ratio between the number of unsatisfied queries for item $j$ traversing peer $i$ and the total number of queries issued in the whole network.

Our objective function is the probability that a query is successful; this is the ratio between the number of successful queries and the total number of queries issued in the network. The problem is first addressed in Section 3 for a static setting where all parameters are fixed. Section 4 shows that our algorithm is efficient in a dynamic P2P network as well. We now have all the ingredients to state our problem formally:

**P2P replication problem**(P2PReplication): Given a P2P network, query rates $q_{ij}$, peer capacities, a set of items together with their corresponding sizes, an assignment of (untouchable) data items to peers, an integer $k$, the goal is to find a feasible assignment of replicas to peers in order to maximize the probability that a query (modeled by the above stochastic process) is successful.

For the sake of presentation clarity, we are assuming all peers to use a random walk with length $k$ as a search protocol, however, our algorithm can be easily adapted to optimize the performance of more sophisticated search protocols, like flooding or multiple random walks, as well as to handle the scenario when peers use different protocols.

### 1.3 Contribution

This paper aims to overcome the limitations of the prior work by addressing the heterogeneity and dynamics of realistic networks and workloads. We have developed an algorithm for dynamic replication coined P2R2 (P2P Replication with 2-Approximation) that maximizes the probability of successfully finding a data item in a query that randomly probes a bounded neighborhood of the query initiator. Random probing means performing length-bounded random walks, which in turn mimic the bounded flooding schemes adopted by real P2P file-sharing systems. Both the objective function and the model for the search procedure are similar to the optimization of [6], but our computational model is more general by allowing arbitrary network topologies and by considering query rates that are specific for each pair of querying node and queried item. Moreover, we not only compute replication degrees for steady state, but also concrete assignments of replicas to nodes and these are continuously adjusted to evolving network and workload properties. A near-optimal assignment is guaranteed for sufficiently long stable periods; load shifts or network changes automatically lead to re-optimization. Our main theorem states that, upon freezing all input parameters for data, load, and network, the distributed algorithm for data replication converges to a $\frac{1}{2+\epsilon}$-approximation of the optimal assignment of replicas to nodes for any $\epsilon > 0$. Experiments with detailed simulations demonstrate the algorithm's excellent behavior even under conditions that go beyond the assumptions of the theorem.

## 2. RELATED WORK

There is a rich body of research results on distributed replication (see, e.g., the survey [21]) and epidemic dissemination, also known as flooding or gossiping (see, e.g., [8]). The work of Cohen and Shenker [6] is one of the first theoretical studies of replication in unstructured P2P networks. It considers a simple search protocol where at each step a peer is selected uniformly at random until the desired data item is found. The main result of the paper is a closed formula for the optimal number of replicas to minimize the expected number of sampled peers. A drawback is that sampling peers independently at random in an arbitrary network is expensive, and ignores the fact that accessing peers which far away in the network is more expensive than accessing neighbors.

Another limitation of [6] is that the optimal number of replicas is expressed as a function of global query rates which, in a practical system, would be unknown to an individual peer; this makes a distributed implementation non-obvious. Solutions have been proposed ([15],[16]), but a practically viable solution is hard because of heterogeneity (of item sizes, peer capacities, and peer-specific query rates) and network dynamics.

Another interesting result is the work of Morselli et al. [17]. It proposes both a search and a replication protocol which achieve exponentially increasing success probability as the number of replicas increases. The protocols involve a random walk followed by a "deterministic" walk which ends in what the authors call "a local minimum for the current item". This ensures that items are placed in or near locations with high request rates. A drawback of this approach is that the number of "local minima" may be large ($O(\frac{n}{\delta})$ in expectation, where $\delta$ is the minimum degree of the graph). If this the case, the network may have not enough storage capacity in order to guarantee the desired probability of success. The result relies on the fact that the distribution of the vertices visited by a random walk with length $t$ (where $t$ depends on the graph) approaches a unique stationary distribution. Unfortunately, there are cases when $t$ has to be large in order for the property to hold, which would in turn result in a large number of messages.

The work of [10] gives theoretical properties of random walks in unstructured P2P networks. Concerning structured P2P networks, we mention EpiChord [14] and Beehive [20], which is a replication framework achieving constant lookup performance.

The literature contains theoretical studies of similar problems; we think that realizing their hardness will help the reader appreciate the simplicity of our solution. We mention the "maximum coverage problem with group budget constraints" [4], the general assignment problem [23], facility location with installation service for which a 6-approximation algorithm has been developed in [22], and the data placement problem for which there is a 20.5-approximation algorithm [1] later improved in ([11]) with a 10−approximation algorithm.

Replication has also been studied as means for higher reliability and availability. Relevant recent work includes the systems projects DHash [7] TotalRecall [2], and Carbonite [5]. All of these are rather pragmatic, without provable guarantees; and they do not specifically consider search.

## 3. THE CENTRALIZED ALGORITHM

The problem defined in the previous section is NP-hard as it can be used to model the 0/1-Knapsack problem. Since an exact polynomial algorithm is unlikely to exist, our effort is to devise an approximation algorithm.

It turns out that the problem is related to the multi-knapsack problem [3]: we are given a set of bins, each with a given capacity, and a set of elements, each with size and profit. We aim to find a set of elements with maximum profit which has a feasible assignment to bins (a feasible "packing").

To see the relation to our P2P replication problem we proceed as follows: each peer is turned into a bin with the same capacity, and for each data items $n$ copies are made, each being an element we wish to place in a bin. The size of each copy of an element $j$ is given by the size of the corresponding data item; the profit we gain by placing a copy of item $j$ in bin $i$ is set to be the number of queries that can be successfully satisfied by placing $j$ in peer $i$ at any given step. The fact that the profit of a given element may depend on the bin the element is placed in as well as on the placement of other items, makes the problem more interesting. Quite surprisingly, we can prove that an approach inspired by the greedy algorithm for multi-Knapsack gives a $\frac{1}{2+\epsilon}$-approximation guarantee for the P2P replication problem.

We now sketch the main operations of our algorithm. In a centralized setting we assume that each peer is aware of all data items stored somewhere in the network and we ignore the problem of how to move data items from one peer to another and learn about their current locations. The distributed P2R2 algorithm presented in Section 4 will drop these idealistic (and actually unrealistic) assumptions.

We start with a configuration where each peer $i$ contains only its natively owned files, which are untouchable. We are making replication decisions for other items to be kept at peer $i$ (for each $i$).

At each step, we *arbitrarily* select a peer $i$, and we compute, for each item $j$, the probability $p_{ij}$ that a query issued by some peer $l$ for item $j$, generates an unsatisfied random walk (of length $k$) traversing peer $i$.

This probability takes the peer-specific query rates $q_{lj}$ into account. In a centralized setting (with complete knowledge of the network and current placement of data items) all $p_{ij}$ values can be computed efficiently in polynomial time, by adapting the standard Chapman-Kolmogorov equations for discrete-time Markov chains [18], to our settings. In a distributed settings, such a procedure would be expensive, and Section 4 will discuss how each peer can compute these probabilities locally and efficiently.

After the $p_{ij}$ values are computed for the peer $i$, we execute an (approximation) algorithm for the knapsack problem with the following input. The peer $i$ is turned into a bin with capacity $c_i$ and each query is turned into an element with size $s_j$ and profit $p_{ij}$. We then select another peer $l$ and compute the $p_{lj}$ values in order to take into account the files placed in other peers. Such procedure is iterated until all peers are processed (in any arbitrary order).

Algorithm 2 shows pseudocode for this centralized replication algorithm, and Algorithm 1 shows pseudocode for a simple greedy algorithm for a generalized version of Knapsack developed in [12].

The following theorem states that, in the case an $\alpha$- approximation algorithm for knapsack is used by each peer, the above procedure has approximation guarantee $\frac{\alpha}{\alpha+1}$.

**Theorem 1** *Given an $\alpha$-approximation algorithm for Knapsack, Algorithm 2 is a $\frac{\alpha}{\alpha+1}$-approximation algorithm for the P2P replication problem.*

Proof. We are given an assignment of untouchable data items to peers. First, we reformulate our problem in order to stress the connections with knapsack and its variants.

Let $Q$ be a set of queries with the following property: for all items $j$ and for all random walks $W$, the ratio between the number of queries for $j$ going through $W$ and $|Q|$, is exactly the probability that a query for $j$ goes through $W$ in our stochastic model. Moreover, we let $R$ be the set of queries in $Q$ which are successful in the sense that they are satisfied by one of the untouchable items. Our problem is then, reformulated as the problem of finding an assignment of data-item replicas to peers in order to satisfy the maximum number of queries in $Q \setminus R$.

It is clear, that the two problems are equivalent, that is, any solution to the "new" problem is $\alpha$-approximated if and only if such a solution is an $\alpha$-approximated solution to the original P2P replication problem (as the value of any solution for the latter problem is always a fraction $1/|Q|$ of the former one). Hence it suffices to prove the claimed approximation guarantee for our reformulated problem.

Peers are assumed, without loss of generality, to have identifiers in the order in which they are processed by our algorithm. Let $O$ be the set of queries satisfied by an optimum solution and let $G$ be the set of queries satisfied by our algorithm. Let $O_i$ be set of queries, traversing peer $i$, which are satisfied by the optimum solution but not by our algorithm. Moreover, let $G_i$ be the set of queries satisfied by peer $i$ in our algorithm and not satisfied by any peer in $\{1, \ldots, i-1\}$. Finally, denote with $Q_{ij} \subseteq Q$ the set of queries for item $j$ which are unsatisfied at step $i$ (i.e., after the choice of replicas for peers $1, 2, \ldots, i-1$ and before making the choice for peer $i$) and which traverse peer $i$.

For each peer $i$ at step $i$, we define the following instance of the knapsack problem. For each element $j = 1, \ldots, m$ we set its size to $s_j$ and its profit to be the cardinality of $Q_{ij}$; the total capacity of the bin is set to $c_i$.

Let $C_i$ be an optimum solution for such problem, for all $i$. We have that $|C_i| \geq |O_i|$, as all elements in $O_i$ are candidates for the optimum knapsack. Remember that the algorithm executed by each peer has approximation guarantee $\alpha$. We then obtain $|G_i| \geq \alpha|C_i|$, as all elements in $\cup_{ij} Q_{ij}$ are candidates to our algorithm and $C_i \subseteq \cup_{ij} Q_{ij}$. Using the two inequalities we derived so far, we have:

$$|G_i| \geq \alpha|C_i| \geq \alpha|O_i|. \qquad (1)$$

By summing up the number of queries satisfied by each peer in our algorithm, we obtain:

$$|G| = \sum_{i=1}^{n} |G_i| \geq \sum_{i=1}^{n} \alpha|O_i|, \qquad (2)$$

where the first equality follows from the definition of the $G_i$'s and the inequality follows from Equation 1.

If $\sum_{i=1}^{n} |O_i| \geq \frac{|O|}{1+\alpha}$ then $|G| \geq \frac{\alpha}{1+\alpha}|O|$, and we are done. Otherwise, by definition of the $O_i$'s, our algorithm will satisfy the remaining $1 - \frac{1}{1+\alpha}$ fraction of the queries satisfied in $O$.

■

The knapsack problem is well-studied, and many algorithms have been developed for it. For our purpose we mention a $(1+\epsilon)$-approximation algorithm with running time $O(n \log \frac{1}{\epsilon} + \frac{1}{\epsilon^4})$ [13], and a simple greedy algorithm with approximation guarantee $(1 - \frac{1}{e}) \approx 0.316$ developed in [12] for a more general case (see also Algorithm 1).

By plugging either one of these results into Theorem 1 we obtain:

**Corollary 1** *Algorithm 2 using the algorithm in [13] as subprocedure is a $\frac{1}{2+\epsilon}$-approximation algorithm for the P2P replication problem, for any $\epsilon > 0$.*

**Corollary 2** *Algorithm 2 using the algorithm in [12] as subprocedure is a 0.24-approximation algorithm for the P2P replication problem.*

The last result has an approximation guarantee that is worse than the previous one, but its simplicity and efficiency make it suitable in a network with high churn (where such algorithm may have to be executed frequently) or with peers having low computation power. Moreover, it can also be used to solve a variant of the P2P replication problem that we will discuss later.

It is worth remarking that all these results hold *regardless* of the order in which peers are processed, as no assumption is made on the ordering. In the next section, we will exploit this property in order to extend the algorithm for distributed processing without any centralized knowledge or control.

**Algorithm 1** $BudgMaxCov(i, p_{i1}, \ldots, p_{im}, c_i, s_1, \ldots, s_m)$

```
 1: $t \leftarrow 0$; $D \leftarrow \{1, \ldots, m\}$; $G \leftarrow \emptyset$;
 2: repeat
 3:    select item $j$, not in $i$, which maximizes $\frac{p_{ij}}{s_j}$;
 4:    if $t + s_j \leq c$ then
 5:       $G \leftarrow G \cup j$;
 6:       $t \leftarrow t + s_j$;
 7:    end if
 8:    $D = D \setminus j$;
 9: until $D = \emptyset$
10: Select an item $l$ in $\{1, \ldots, m\}$ which maximizes $p_{il}$;
11: If $\sum_{j \in G} p_{ij} \geq p_{il}$, then return $G$, otherwise return $l$;
```

**Algorithm 2** Optimize

```
1: $Q \leftarrow \emptyset$;
2: let $v_1, \ldots, v_n$ be any arbitrary ordering of all peers;
3: for i=1, ..., n do
4:    compute/update $p_{i1}, \ldots, p_{im}$;
5:    let $\mathcal{A}$ be an $\alpha$-approximation algorithm for the knapsack problem;
6:    $G_i \leftarrow \mathcal{A}(i, p_{i1} \ldots p_{im}, c_i, s_1, \ldots, s_m)$;
7:    let peer $i$ store all items in set $G_i$;
8: end for
```

## 4. THE DISTRIBUTED ALGORITHM

In this section we present our main result: the fully decentralized P2R2 algorithm for the P2P replication problem. The distributed algorithm mimics the centralized one in an efficient way. It turns out that this is non-trivial as we want to achieve two conflicting tasks. First, each peer should take into account replicas placed by other peers in their neighborhood; otherwise the most popular items may be replicated everywhere even if this is not necessary. On the other hand, we wish to guarantee a certain level of independence and parallelism for the replication decisions of different peers. This problem is made even more complicated by the limited knowledge each peer has about the network as well as the dynamics of the P2P network, where both query rates and the structure of the network may rapidly change.

We characterize the performance of our P2R2 algorithm in *steady* state, where a steady state is defined as the period of time throughout which the graph structure of the network, the query rates, and the set of untouchables items for each peer do not change. We assume the number of *steady* states to be constant. Our algorithm will be continuously running in a dynamic P2P network, reconsidering and revising the replication decisions. In any *steady state*, the approximation guarantee of our distributed algorithm will converge to the guarantee achieved by the centralized algorithm of the previous section. As the convergence is guaranteed in *any* steady state, this implies that, when the network changes because of peer churn or the query rates change, our algorithm will automatically adapt the replication to the new setting.

## 4.1 Algorithm

For presentation clarity, we first assume that the diameter of the network graph is smaller than the length of any random walk. This ensures, that any peer is potentially reachable by any other peer in the network. We will refer to a graph whose diameter is at most the length of any random walk as a *graph with small diameter*. In Subsection 4.3, we show how to carry over the algorithm to the situation with large diameter graphs.

Moreover, we assume that each peer and each query are identified by an *id*. The *id* of a peer may be its IP number, and the id of a query may be, for example, the time the queries is issued together with the IP of the query initiator. We assume peer *id*'s to be $1, \ldots, n$ and item *id*'s to be $1, \ldots, m$.

The key problem which we need to solve in order to turn the centralized algorithm into a distributed one, is how to compute the probabilities $p_{ij}$ for each peer $i$ and item $j$. Recall that this is the probability that an unsatisfied query for item $j$ traverses peer $i$. These probabilities have a global nature and cannot be efficiently computed in a distributed setting. This is why we introduce a new measure, the $r_{ij}$ values, which replace the $p_{ij}$ values and serve to estimate the $p_{ij}$ values. For each peer $i$ and each item $j$, we denote with $r_{ij}$ the number of queries for item $j$ issued until the current time such that: 1) random walks generated by the query traversed peer $i$, and 2) either the query is not satisfied (i.e., is unsuccessful) or it is satisfied by a peer with *id* larger than $i$.

The quantities $r_{ij}$ are computed as requests are disseminated in the network. To this end, each peer that receives, and possibly further forwards, a query request needs to maintain appropriate per-item counters. We remark that this is the only bookkeeping required, which is clearly low.

What we additionally need is a mechanism to inform all peers in any random walk, whether a given query is eventually satisfied or not. For this reason, the distributed algorithm requires some additional, but small overhead: each request is delivered by a random walk with maximum length $k$, and the answer is sent back to the peer who issued the query along the same path. So the peer that holds the item and provides the answer does not reply directly to the query originator. Moreover, the random walk is not stopped, and we thus have the full path length $t$, even if the requested data item is found earlier. Clearly, this incurs acceptable extra message costs, and our experiments confirm this.

When spreading a peer's request in the network, each random walk delivers a message with following fields:

1. The field *queryInitiator*, with the *id* of the peer issuing the query.
2. The field *item* storing the *id* of the item which the query has been issued for.
3. The time to live (TTL) counter. If this field has value one, (which is not the case when the message is going forward) it indicates an answer message is going backward to the query originator.

Additional fields are necessary to compute the $r_{ij}$ values, namely:

4. The flag *satisfied* specifying whether the query is successful or not.
5. The field *satBy* whose value is relevant in the case the query is successful. It contains the *id* of the peer whose *id* is minimum among the ones hold item $j$ in the current random walk.

We now describe the main operations executed by the distributed P2R2 algorithm. Each peer $i$ locally computes quantities $r_{ij}$ for each item $j$ that the peer is aware of. These quantities are initially set to zero and updated each time a peer is traversed by a random walk. As the message goes through the backward path of the random walk, the *satisfied* and *satBy* fields are updated. For these updates, each peer $l$ receiving such a message checks both the values of *satisfied* and *satBy*. If *satisfied* is set to true or *satBy* is larger than or equal to $l$ then $r_{il}$ is increased by one.

Each $r_{ij}$ value measures the current *worthiness* of item $j$ for peer $i$. In order to compute which items have to be stored in $i$, each peer executes any algorithm for knapsack with the following input. The set of items consists of all items for which the peer knows the location of one replica (and which are not untouchable); for each such item $j$, the algorithm receives as input also the value $r_{ij}$ (instead of $p_{ij}$). We observe that this knapsack-computation procedure may delete some non-worthy data items previously stored at the peer, to account better estimates of the $p_{ij}$'s values or changes in the network.

There may be *worthy* items for which no replica location is known to the peer under consideration (this happens when such items were never found in any prior query by this peer). For each peer $i$ we refer to an item $j$ as *missing* if the value $r_{ij}/s_j$ is larger than the value of at least one item in the knapsack solution of peer $i$. Each peer aims to find *missing* items, as additional replicas of these items may improve the global performance (i.e., query success probability). For this purpose, the following two fields are included in each message, effectively piggybacking the inquiries about missing items on the regular messages:

6. The field *storedItem* with the *id* of one (or more) randomly chosen untouchable item(s) stored by the query initiator.

7. The field *missingItem* with the *id* of one (or more) randomly chosen missing item(s) of the query initiator.

So each time a peer $i$ issues a query it also pushes information about one (or more) of its untouchable items and one (or more) of the missing items it knows about. The former tells other nearby peers that there is already one copy of this item "in the neighborhood" (namely, the untouchable, original copy); the latter tells other peers that there is a demand for having a replica of the missing item "in this neighborhood".

In our distributed algorithm, a peer replies to a missing-item request (piggybacked on a regular search for some other item) only if the missing item is untouchable in that peer (i.e., an original item rather than a derived replica). This choice is purely dictated by convenience for the analysis of the algorithm's approximation guarantee. In practice, peers may also report about non-original replicas, but we may then have to consider issues of derived replicas becoming stale - this aspect is orthogonal to our approach. The simplification for the analysis is that we can study the algorithm in steady state when there is a probability strictly larger than zero that any peer finds any missing item. It would be interesting to experimentally evaluate modifications of the algorithm along these lines, which are hard to characterize by mathematical analysis.

This concludes the description of our algorithm; Algorithm 3 shows pseudocode for the main operations executed by each peer when receiving a query.

## 4.2 Approximate Guarantee

We start by giving an intuition for why the $r_{ij}$ values are defined in the particular way given above. The $r_{ij}$ values explicitly ignore the data stored in peers whose *id* is larger than the current peer *id* $i$. This allows the distributed algorithm to deal with the possibility that each peer computes a partial assignment of data items at the very same time. Obviously, such totally unsynchronized parallelism may mislead the decisions of the peers. For example, neighboring peers may replicate the same item although it would be good enough to have only replica in this same neighborhood. We want to solve this need for "coordination" without introducing any explicit synchrony among peers. This is exactly what the condition about peer *id*'s in the updates to the $r_{ij}$ values achieves. To see this, think about the distributed algorithm as mimicking the centralized one, in which at any given step $i$, no assignment is computed by peers whose *id* is larger than $i$. If we synchronized the placement decisions among the peers by enforcing such an explicit order, we would avoid the above pitfalls. But this strong form of synchrony would come at a high cost and would seem inappropriate for a loosely coupled P2P network. Our solution allows peers to proceed in a fully asynchronous manner but achieves the same effect.

This approach guarantees a good level of parallelism, as at any given time, all peers are working at improving the performances of the network; moreover we can expect our algorithm to rapidly converge to a good approximated solution since in many cases peers that are sufficiently far apart may indeed make their decisions independently.

The $r_{ij}$ values are substitutes for the $p_{ij}$ in the centralized algorithm. In fact, it can be shown by using the strong law of large numbers together with induction, that for each $i$ and $j$, $r_{ij}$ converges to $p_{ij}$ multiplied by the total number of queries issued in the network. Now it is clear that we obtain the same performance guarantee as in the centralized algorithm, since profits in these two problems are the same, apart from a multiplicative factor which is the total number of queries issued in the network. The next theorem states the approximation guarantee of the distributed P2R2 algorithm in the case when the $(1+\epsilon)$-approximation algorithm from [13] is used as a subprocedure.

**Theorem 2** *In any sufficiently long steady state with a small diameter graph, the distributed P2R2 algorithm for P2P replication converges to a $\frac{1}{2+\epsilon}$-approximation solution, with high probability (for any $\epsilon > 0$).*

Proof. Consider the state of the P2P network at any time $t$. Let $Q_t$ be the set of queries issued according to our model until such time. We prove the claim in the case when there is a sufficiently large integer $\bar{t}$, such that the P2P network is *steady* in the time interval $[t, \bar{t}]$.

The proof uses the Chernoff's bound and induction on the number of peers, in order to show that, for all $i$ ($i = 1, \ldots, n$), the solution computed by the distributed algorithm for the subproblem at the first $i$ peers, matches a centralized solution for the corresponding subproblem.

Let $\bar{r}_{i,j}$ be the value of $r_{i,j}$ at time $\bar{t}$. In the case $i = 1$, $\bar{r}_{1,j}$ counts the number of queries traversing peer 1, as the content of all other peers is ignored. By Chernoff's bound, if $\bar{t}$ is sufficiently large, then $\frac{\bar{r}_{1,j}}{|Q_{\bar{t}}|}$ is sufficiently close to $p_{1,j}$ for all $j$ $j = 1, \ldots, m$, with high probability.

Moreover, since requests for *missing* items are propagated in the network, there is a constant probability $\delta > 0$ that peer 1 finds out the location of all missing items that it knows of. Let $n_t$ be the number of requests of peer 1 up to time $t$. Then the following holds:

$$\mathcal{P}(\textit{peer } 1 \textit{ finds his mis. items at time } t) = 1-(1-\delta)^{n_t} \quad (3)$$

which converges to 1. Therefore, peer 1 finds all missing items with high probability and the set of items stored in his location matches a centralized solution with the claimed approximation guarantee.

For $i \geq 2$, peer $i$ ignores only the content of peers whose *id* is larger than $i$. This simulates the scenario of the centralized algorithm, where no data item is assigned to any such peers. By the inductive hypothesis, there is a centralized solution which matches exactly the distributed one for the first $i-1$ peers. Moreover, by using the same argument used above, we can prove that peer $i$ finds all its missing items with high probability. It follows that $\frac{\bar{r}_{1,j}}{|Q_{\bar{t}}|}$ is a good approximation of $p_{ij}$ and the two solutions coincide up to the first $i$ peers, with high probability.

To conclude the proof, we use the union bound over all the constant number of steady states. ■

## 4.3 Large Diameter Graphs

If the P2P network has a diameter larger than the maximum length of any random walk, we need a mechanism to propagate the information about the *missing items* without flooding the network. One way to do this, is to let the query initiator append to its queries, a request for items which are *missing* by *other peers*. This way we can disseminate information about missing items beyond the reach of any individual random walk, but we still piggyback all of

**Algorithm 3** Distributed Algorithm: Receive

```
1: for each peer $i$ in parallel do
2:   while true do
3:     if $i$ receives query $q$ for item $j$ and $TTL > 1$ (the message is going forward) then
4:       if $i$ contains $j$ then
5:         If $j$ is untouchable then
             $q.satisfied \leftarrow true$; $q.satBy \leftarrow -1$;
6:         ElseIf $q.satisfied = false$ then
             $q.satisfied \leftarrow true$; $q.satBy \leftarrow i$;
7:         ElseIf $q.satisfied = true$ then
             $q.satBy \leftarrow min(q.satBy, i)$;
8:       end if
9:       $TTL \leftarrow TTL - 1$;
10:      If $TTL > 1$ then forward the message to a random neighbour;
11:      If $TTL = 1$ then send the message backward;
12:    end if
13:    if $i$ receives query $q$ for item $j$ and $TTL = 1$ (the message is going backward) then
14:      If $q$ is the first query for item $j$ then $r_{ij} \leftarrow 0$;
15:      If $q.satisfied = false$ OR $q.satBy \geq i$ then $r_{ij} \leftarrow r_{ij} + 1$;
16:      If $i \neq q.queryInitiator$ then forward the message backward;
17:    end if
18:    let $1, \ldots, p$ index all items that peer $i$ is aware of (and which are not untouchable);
19:    let $\mathcal{A}$ be an $\alpha$-approximation algorithm for the knapsack problem;
20:    $G_i \leftarrow \mathcal{A}(i, r_{i1}, \ldots, r_{ip}, c_i, s_1, \ldots, s_p)$;
21:  end while
22: end for
```

this dissemination on the regular message traffic that takes place on behalf of actual queries.

The following additional fields are required:

8. The field *missingItemByOthers* with the *id* of one (or more) items missing by other peers, which the query initiator knows about.

9. The field *missingBy* which contains the *id* (or a list of ids) of the peer (or peers) missing the *missingItemByOthers*.

Each request for *missingItems* has a probability $\delta > 0$ to reach (hop by hop) a peer containing such an item. Thus, by the arguments used in the proof of Theorem 3, we are able to generalize our result to any arbitrary network.

**Theorem 3** *In any sufficiently long steady state, the distributed P2R2 algorithm for P2P replication converges to a $\frac{1}{2+\epsilon}$-approximation solution, with high probability (for any $\epsilon > 0$).*

## 5. IMPLEMENTATION ISSUES

This section discusses implementation issues for distributed replication algorithms, aiming to bridge the gap between the computational models in the literature and practical P2P systems.

The P2R2 algorithm itself can be implemented relatively easily as shown in the pseudo code. The only issue is the approximation algorithm for the knapsack problem, which is invoked very frequently and could be expensive. When using Algorithm 1 for approximating the Knapsack solution (with $1 - \frac{1}{e}$ approximation guarantee, but typically even closer to the optimum), items are kept in a Pairing heap [9] (or a Fibonacci heap), sorted according to their worthiness ($\frac{r_{ij}}{s_j}$). When a new request arrives, the worthiness increases, which is an $O(1)$ operation for Pairing heaps. When the new worthiness of a non-replicated is higher than that of currently replicated objects, the algorithm is run. It only examines the items with the highest worthiness and accesses them in sorted order. When removing them for examination, they be placed on a totally ordered Pairing heap, which can afterwards be re-merged with the main heap in $O(1)$. This makes repeated executions of Algorithm 1 (and thus of P2R2) very efficient.

While implementing other existing methods as competitors for our evaluation, we noticed that some of them are underspecified or hard to use in a real-world situation. For example, Square-root Replication by Cohen and Shenker [6] is initially described as a centralized algorithm, which is not directly applicable in a P2P setting. The authors provide a sketch for a decentralized algorithm that works as follows. Instead of replicating according to the query rates (which are not known from the local viewpoint of a individual peer), the algorithm creates replicas proportionally to the length of the random walk required to find the data item. When randomly chosen replicas are also deleted with a suitable rate, this method converges to the original square-root strategy.

Unfortunately this is underspecified for an actual implementation. How many replicas should be created? The algorithm only gives the proportion, not an absolute number (which would depend on global information). Where should the replicas be created? How should the deletions be scheduled? A follow-up paper [16] explains that the deletion rate must match the creation rate of new replicas, and that deletions should be randomized or follow a FIFO strategy, but it is still not obvious at all how exactly this should be implemented.

Some of these issues have been addressed by Leontiadis et al. in [15], proposing the *Pull-Then-Push* strategy for replication. When a requested item is found after a random walk of length $t$ from the query initiator, a new replica is created and placed by performing another random walk of length $t$ starting from the query initiator and placing the replica at the end point of the walk.

In our experiments, we used this strategy for replica placement, and computed the number of replicas and the deletion rate as follows: when an item $i$ is found after a random walk of length $t$, a new replica is created with probability

$$Prob(\text{replicate } i) = \frac{t}{\text{maximum random walk length}}.$$

This creates replicas in proportion to $t$, as desired by the square-root algorithm. When a replica is created, another random walk of length $t$ is performed (Pull-Then-Push). If the target node already contains item $i$, no replica is created, otherwise item $i$ is placed at the target node. If there is not enough free space available at the node, replicas of other items are deleted at random until item $i$ can be placed. This technique results in a fully decentralized algorithm that converges to square-root replication.

Another prominent competitor is the LMS algorithm [17] which has several implementation issues. First, it has a number of tunable parameters, and it is not clear at all how to choose them. More critically, the LMS algorithm makes its replication decision based on the *number of reachable repli-*

*cas* $r$, which is computed as a smoothed average of $s$, the number of probes required to find an item. It is unclear how to compute an appropriate value for $r$ in a distributed setting without centralized control. The individual $s$ values per item are known by the query-initiating peers, but they cannot be easily aggregated over the entire network. Such aggregation is conceivable if the search is successful (assuming that the replicas organize a distributed counter), but a failed search does not reach any node with information about the specific data item. Averaging only over successful searches leads to very poor performance, as the required number of probes would then be systematically underestimated (the algorithm would then even deletes replicas although more replicas should be allocated). In our experimental studies, we assumed that these statistics are available and accurate, although this is most likely not feasible (at affordable cost) in an unstructured P2P network. Thus, we are giving the algorithm (as an opponent to ours) the advantage of some centralized information.

We implemented the LMS algorithm as follows. The probes are executed as described in [17]. The statistics on the required number of probes are maintained in a centralized manner for both successful and unsuccessful searches. (This actually gives the algorithm an advantage that a truly decentralized implementation would not have.) Upon a successful search, replicas are created or deleted as described in Section 3.4 of [17]. As for tunable parameters, the LMS algorithm needs to set the desired number of replicas, the maximum number of probes, and the maximum random walk length (the deterministic walk is unbounded). For broader and systematic experimentation, we preferred specifying only the maximum number of networks hops $k$, and then derived the LMS parameters as follows:

$$\begin{aligned}\text{desired \#replica} &= \frac{\sum_i c_i}{\sum_j s_j}\\ \text{max \#probes} &= \lfloor\sqrt{k}\rfloor\\ \text{max random walk length} &= \lceil\frac{\sqrt{k}}{2}\rceil\end{aligned}$$

The desired number of replicas is chosen according to the available capacity, and the number of networks hops is distributed evenly in both dimensions (number of probes vs. walk length). The random walk length in only half of the available hops, as the other half is expected to be used by the deterministic walk. We performed extensive experiments and found that these parameters work fairly well.

## 6. EVALUATION

### 6.1 Setup

In addition to the theoretical analysis of the approximation quality that the P2R2 algorithm achieves, we were interested in studying its practical behavior in situations that go beyond our computational model, most importantly, when the network or the workload undergo transient phases (i.e., when the system is not in steady state). In this section we provide experimental results based on a detailed simulation testbed. We compare P2R2 to the state-of-the-art competitors square-root replication (including the Pull-Then-Push enhancements from [15]) and LMS search, with the implementation techniques described in Section 5.

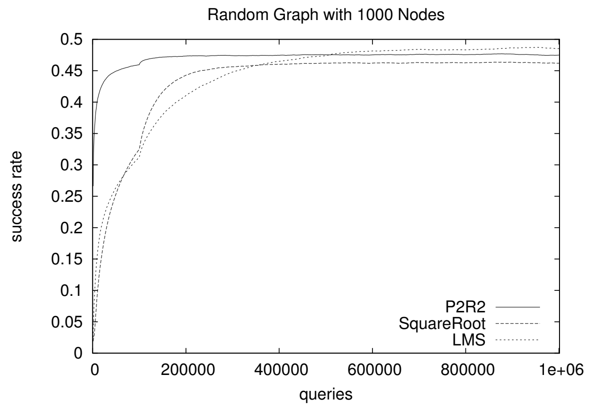

**Figure 1: Results for simple random graphs with edge density 0.02**

We report on two lines of experimentation: one with a synthetically generated workload (with skewed query rates) and one based on a real-life query log from the Gnutella P2P network. In each experiment we create a certain network topology and workload profile, and then execute 1 million queries. With the synthetic load, we first choose the query initiator uniformly at random and then choose the queried item based on the chosen peer's specific query rates. With the real-life load, the query log determines both the initiator and the queried item. We discuss both the overall performance (success rate) in steady state as well as the convergence rate after bursts of network churn (peers leaving and new peers joining).

### 6.2 Results

*Impact of Topology*

We first study the behavior of the different algorithms on different network topologies, using a synthetic workload. For each experiment we create a specific topology (discussed below), place 5 items of size 1 on each peer (with a peer capacity of 10), and a Zipf-distributed query load with $z = 1$ (i.e., with the probability of requesting the $j^{th}$ most popular item being proportional to $(\frac{1}{j})^z$). We set the maximum path length of the random walk to 20. We present results for the average success rate as a sliding average over 100,000 queries.

As one of the simplest topologies, we study random graphs where nodes are connected with a certain probability. Figure 1 shows the results for random graphs with 1000 nodes and an edge probability of 0.02. The figures are averages over 10 random graphs. All three algorithms perform nearly identical in this simple case. LMS performs slightly better, but depends on global information (see Section 5). When using only locally available information, it performs much worse. An interesting observation is that the convergence rates differ widely. P2R2 is by far the fastest converging method, and square-root replication converges faster than LMS.

A consequence of this can be seen in Figure 2. It shows the same setup as in the previous figure, but after every 200,000 queries 80% of the peers randomly leave the system, with an equal number of new peers joining. In the charts, the sliding window size is reduced to 10,000 queries to show the churn

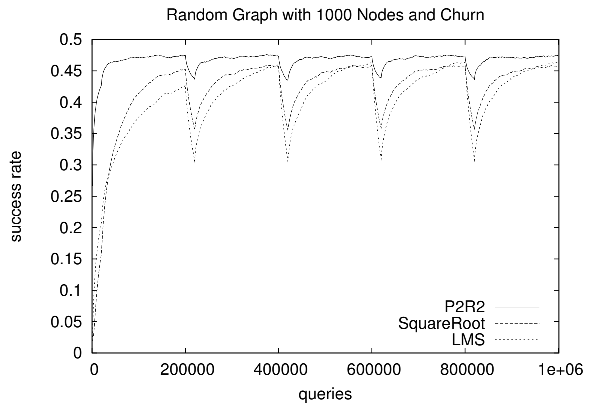

**Figure 2: Results for same graph as in Figure 1, but with churn every 200,000 queries**

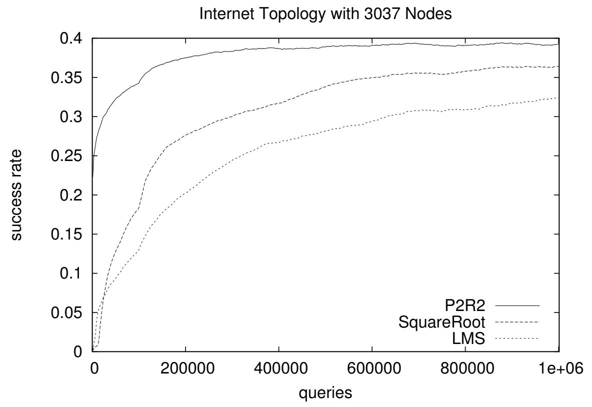

**Figure 3: Results for Internet-style topology**

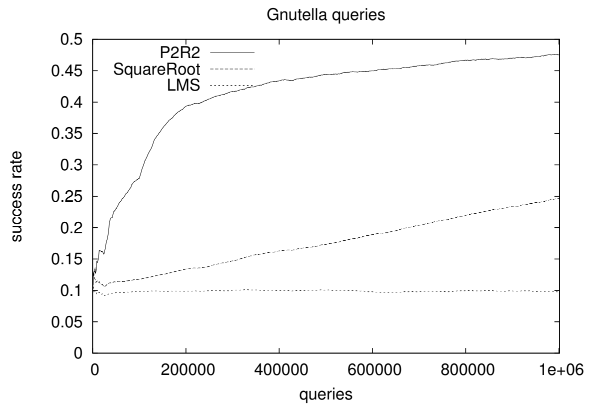

**Figure 4: Results for Gnutella queries**

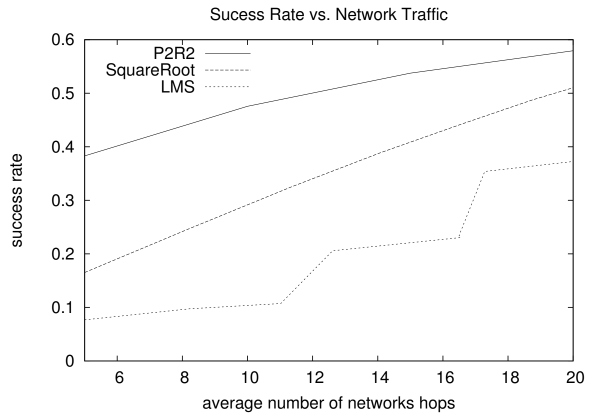

**Figure 5: Results for the same setup as Figure 4 but with varying network traffic**

effect in more detail. Our P2R2 algorithm handles this situation very well, with quick "recovery" after the churn bursts. The other two algorithms are more severely affected, with a sharp drop of their success rate after each of the major network changes. For LMS this effect is even more pronounced, as it converges slower than square-root replication.

In addition to simple randoms graphs, we studied Internet-style topologies as generated by Inet-3 [25]. These graphs are more structured than random graphs, with a backbone structure and other realistic features of the Internet. The results for a topology with 3037 nodes are shown in Figure 3. Here the differences between the algorithms are larger, with P2R2 performing best and LMS performing worst. Again, the convergence rates of square-root replication and LMS are much worse than that of P2R2.

We studied even more structured topologies, all the way to a linear chain of nodes, an extremely rigid structure and a kind of stress test. The results are similar, with LMS performing worse the more rigid the structure is. Square-root replication performs reasonably well even for chains, but primarily because of the Pull-then-Push enhancement from [15].

## 6.3 Impact of Real-Life Query Rates

The experiments so far assume that the item popularities follow a Zipf distribution. While this is plausible, it is not necessarily true in practice. The experiments presented here use a real-life query log to drive the simulations. We use the publicly available IR-Wire data set [19], which consists of query logs and 50 peer descriptions from the Gnutella P2P network. We construct a random graph for the 50 peers (edge probability 0.02), assign the items according to the peer descriptions (assuming that each peer fills its storage to 50%), and derive item requests from the query log by interpreting each query as a conjunctive keyword query on the file name. This results in 100,123 queries. We repeat this query log 10 times to study the convergence of the algorithms. The maximum random walk length is set to 10.

The results are shown in Figure 4. P2R2 clearly outperforms the other algorithms, with LMS performing poorly and square-root replication converging very slowly. LMS and square-root replication cannot cope with the query log very well, which contains both heavily skewed queries (i.e., extremely popular items), and many "singleton" queries that occur only once or twice. The latter tend to pollute the replication decisions of the square-root and LMS algorithms: the items are replicated although there is rarely any subsequent benefit. This shows the limitations of these methods in coping with non-stationary conditions. Our P2R2 method avoids this pitfall by computing the replication based on the overall peer statistics instead of individual queries.

We also study the success rates of the different algorithms relative to their network traffic costs. While all algorithms use the same maximum number of hops, some algorithms stop earlier when they find an item. We therefore vary the maximum number of hops, and measure the actual number of hops performed by the different algorithms. Figure 5 shows the success rate vs. the average number of random-walk hops for the previous experiment (both measured after 1 million queries). Again P2R2 performs very well, offering a much better success rate relative to the network traffic costs.

## 7. CONCLUSION

We have presented the P2R2 algorithm for P2P replication, along with theorems about its near-optimal approximation guarantees. In contrast to prior work on replication in unstructured P2P networks, our algorithm is particularly well equipped for coping with heterogeneity and dynamics in the network and workload. Our experimental evaluation, which studied also the transient behavior after bursts of churn (going beyond the steady-state situation that underlies the computational model of the theorems), clearly demonstrates the superiority of the P2R2 algorithm compared to prior state-of-the-art methods.

Our future work includes further extensions and generalizations of the algorithm. Using replication to boost the success probability of range queries and partial-match search is one of the major avenues that we want to explore in more depth. Another dimension to address is to introduce peer-specific load capacities in addition to the space capacities that we already consider; this limits the sustainable throughput for each peer and thus the overall network performance and success probabilities. This calls for significant extensions to the algorithm and its theoretical analysis, and also raises interesting practical issues.

**Acknowledgments** We would like to thank Alessandro Panconesi for helpful suggestions and stimulating discussions.

## 8. REFERENCES

[1] I. D. Baev and R. Rajaraman. Approximation algorithms for data placement in arbitrary networks. In *SODA*, pages 661–670, 2001.

[2] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *NSDI*, pages 337–350, 2004.

[3] C. Chekuri and S. Khanna. A polynomial time approximation scheme for the multiple knapsack problem. *SIAM J. Comput.*, 35(3):713–728, 2005.

[4] C. Chekuri and A. Kumar. Maximum coverage problem with group budget constraints and applications. In *APPROX-RANDOM*, pages 72–83, 2004.

[5] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[6] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *SIGCOMM*, pages 177–190, 2002.

[7] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a dht for low latency and high throughput. In *NSDI*, pages 85–98, 2004.

[8] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67, 2004.

[9] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[10] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks: Algorithms and evaluation. *Perform. Eval.*, 63(3):241–263, 2006.

[11] S. Guha and K. Munagala. Improved algorithms for the data placement problem. In *SODA*, pages 106–107, 2002.

[12] S. Khuller, A. Moss, and J. Naor. The budgeted maximum coverage problem. *Inf. Process. Lett.*, 70(1):39–45, 1999.

[13] E. L. Lawler. Fast approximation algorithms for knapsack problems. *Math. Oper. Res.*, 4(4):339–356, 1979.

[14] B. Leong, B. Liskov, and E. D. Demaine. Epichord: Parallelizing the chord lookup algorithm with reactive routing state management. *Computer Communications*, 29(9):1243–1259, 2006.

[15] E. Leontiadis, V. V. Dimakopoulos, and E. Pitoura. Creating and maintaining replicas in unstructured peer-to-peer systems. In *Euro-Par*, pages 1015–1025, 2006.

[16] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS*, pages 84–95, 2002.

[17] R. Morselli, B. Bhattacharjee, A. Srinivasan, and M. A. Marsh. Efficient lookup on unstructured topologies. In *PODC*, pages 77–86, 2005.

[18] R. Nelson. *Probability, Stocastic Processes, and Queuing Theory - The Mathematics of Computer Performance Modeling*. Springer, 1995.

[19] L. T. Nguyen, W. G. Yee, D. Jia, and O. Frieder. A tool for information retrieval research in peer-to-peer file sharing systems. http://ir.iit.edu/w̃aigen/proj/pirs/irwire/. In *ICDE*, pages 1525–1526, 2007.

[20] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI*, pages 99–112, 2004.

[21] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[22] D. B. Shmoys, C. Swamy, and R. Levi. Facility location with service installation costs. In *SODA*, pages 1088–1097, 2004.

[23] D. B. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Math. Program.*, 62:461–474, 1993.

[24] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[25] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. http://topology.eecs.umich.edu/inet/. Technical Report UM-CSE-TR-456-02, EECS, University of Michigan, 2002.