P-Ring: An Efficient and Robust P2P Range Index Structure*

Adina Crainiceanu
United States Naval Academy
adina@usna.edu

Prakash Linga
Cornell University
linga@cs.cornell.edu

Ashwin Machanavajjhala
Cornell University
mvnak@cs.cornell.edu

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Jayavel Shanmugasundaram†
Yahoo! Research
jaishan@yahoo-inc.com

ABSTRACT

Peer-to-peer systems have emerged as a robust, scalable and decentralized way to share and publish data. In this paper, we propose P-Ring, a new P2P index structure that supports both equality and range queries. P-Ring is fault-tolerant, provides logarithmic search performance even for highly skewed data distributions and efficiently supports large sets of data items per peer. We experimentally evaluate P-Ring using both simulations and a real distributed deployment on PlanetLab, and we compare its performance with Skip Graphs, Online Balancing and Chord.

Categories and Subject Descriptors: H.2.2 Physical Design: Access Methods, H.2.4 Systems - Distributed Databases

General Terms: Algorithms, Management, Performance.

Keywords: peer-to-peer systems, range queries, load balancing.

1. INTRODUCTION

Peer-to-peer (P2P) systems have emerged as a new paradigm for structuring large-scale distributed systems. Their key advantages are their scalability, due to resource-sharing among cooperating peers, their fault-tolerance, due to the symmetrical nature of peers, and their robustness, due to self-reorganization after failures. Due to the above advantages, P2P systems have made inroads for content distribution and service discovery applications [2, 19, 21, 23]. One of the requirements of such systems is to support range queries. For example, in a large computing grid, where each node advertises its resources, one might need to find all the nodes in the grid with enough main memory for a memory intensive application: "Select * From AllNodes M Where M.Memory > 2GB".

†Research done while at Cornell University.

*This material is based upon work supported by the National Science Foundation under Grant 0133481, by the Air Force under Grant AFOSR F49620-02-1-0233, and by Naval Academy under a NARC grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

Copyright 2007 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SIGMOD'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

While there have been several promising P2P range index structures that have been proposed in the literature, they have certain limitations. Skip Graphs [3] and P-trees [5] can only handle a single data item per peer, and hence, are not well-suited for large data sets. The index proposed by Gupta et al. [12] only provides approximate answers to range queries, and can miss results. Mercury [4] and P-Grid [2, 10] provide probabilistic (as opposed to absolute) guarantees on search and load-balancing, even when the P2P system is fully consistent. BATON [14] only provides search performance proportional to $log_2 P$, where P is the number of peers; when P is large, the small base of the logarithm can lead to excessive search cost. BATON* [13] provides search performance proportional to $log_d P$, but it does not prove any guarantees on load balancing.

We propose P-Ring, a new P2P range index. P-Ring provides exact answers to range queries on arbitrary ordered domains, and scales to a large number of peers and data items. P-Ring provides provable guarantees on load-balancing, with load imbalance factor of at most 2+ϵ, for any given $\epsilon > 0$. P-Ring provides search performance of $O(log_d P)$, where P is the number of peers in the system, and d is a tunable parameter. We are not aware of any other P2P index that provides the same functionality and performance.

When designing P-Ring we were faced with two challenges. First, the data items have to be distributed among peers such that range queries can be answered efficiently, while still ensuring that all peers have roughly the same number of data items (for load balance). Techniques developed for equality queries are not applicable as they distribute data items based on their hash value; since hashing destroys the order of the items, range queries cannot be answered efficiently. We need to devise a scheme that clusters data items by their data value, and balances the number of items per peer, even in the presence of highly skewed insertions and deletions. Our first contribution is a scheme that provably maintains a load imbalance of at most 2+ϵ (for any given ϵ >0) between any two peers in the system, while achieving amortized constant cost per insertion and deletion. This achieves a better load balance factor when compared to that of 4.24 proposed by Ganesan et al. [11], while keeping the amortized insert/delete cost constant.

Our second challenge was to devise a query router that is robust to peer failures and provides logarithmic search performance even in the presence of skewed data distributions. Our P-Ring router, called Hierarchical Ring (HR), is highly fault-tolerant, and a router of order d provides guaranteed $O(log_d P + m)$ range search performance in a stable system with P peers, where m is the number of peers with data items in the query range. Even in the presence of highly skewed insertions, we can guarantee a worst-case search

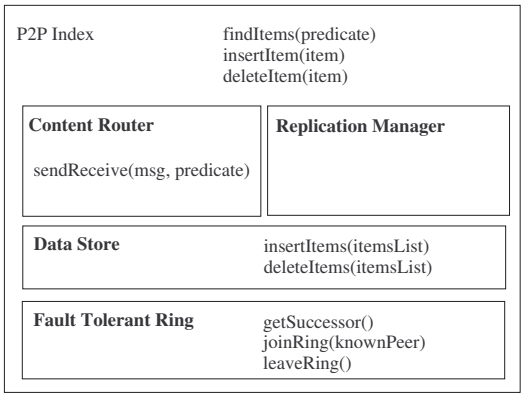

Figure 1: P2P Framework

cost of $O(r \cdot d \cdot log_d P + m)$, where r is the number of peer insertions per stabilization unit of the router (we will formally define all terms later).

In a simulation study, we compare the performance of P-Ring to an extension of Skip Graphs [3] and to Chord[23]. Our performance results indicate that P-Ring outperforms the above extension of Skip Graphs in terms of both query and update cost. P-Ring offers the same (if order d of HR is 2) or better (if $d > 2$) search performance than Chord, but at a higher cost, due to the support of additional functionality (range queries as opposed to only equality queries). We also present preliminary experimental results from a real distributed implementation of P-Ring, Chord and Online Balancing [11] deployed on PlanetLab [1], a network of computers distributed around the world.

2. MODEL AND ARCHITECTURE

System Model. We assume that each peer in the system can be identified by an address (IP address and port number), and peers can communicate through messages. A peer can join a P2P system by contacting some peer that is already part of the system. A peer can leave the system at any time without contacting any other peer. We assume that while in the system, the peers follow the distributed indexing protocol. This assumption is consistent with other papers in the literature [11, 14, 23]. In this paper we use P to denote the number of peers in the system.

We assume that each data item (or short, item) stored in a peer exposes a *search key value* from a totally ordered domain that is indexed by the system. Without loss of generality, we assume that search key values are unique. Duplicate values can transparently be made unique by appending the address of the peer where the value originates and a version number. We use N to denote the number of items in the system.

For simplicity, we assume that the query distribution is uniform, so the load of a peer is determined by the number of data items stored at the peer. The algorithms introduced in this paper work with any definition of the load.

We define the *load imbalance* in a system to be the ratio between the most loaded and the least loaded peer in the system.

System Architecture. We have implemented P-Ring in an architecture similar to the modular framework of [6]. We now overview the relevant components of the framework (Figure 1).

Fault Tolerant Ring: The Fault Tolerant Ring connects the peers in the system along a ring, and provides reliable connectivity among these peers even in the face of peer failures. For a peer p, we can define the $\texttt{succ}(p)$ (respectively, $\texttt{pred}$(p)) to be the peer adjacent to p in a clockwise (resp., counter-clockwise) traversal of the ring. Figure 2 shows an example of a Fault Tolerant Ring. If peer p_1 fails, the ring will reorganize such that $\texttt{succ}(p_5) = p_2$, so the peers remain connected. Figure 1 shows the Ring API. The Ring provides methods to get the address of the successor, join the ring or gracefully leave the ring (of course, a peer can leave the ring without calling `leaveRing` due to a failure). In our implementation of P-Ring, we use Chord's Fault Tolerant Ring [23].

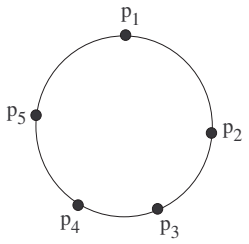

Figure 2: Fault Tolerant Ring

Data Store: The Data Store is responsible for distributing the items to peers. Ideally, the distribution should be uniform so that each peer stores about the same number of items, achieving storage balance. One of the main contributions of this paper is a new Data Store for P-Ring, which can effectively distribute items even under skewed insertions and deletions (see Section 3).

Content Router: The Content Router is responsible for efficiently routing messages to peers that have items satisfying a given predicate. The second major contribution of this paper is a new Content Router that can route range queries efficiently (see Section 4).

Replication Manager: The Replication Manager ensures that items assigned to a peer are not lost if that peer fails. We use the Replication Manager proposed in CFS [8].

P2P Index: The P2P Index is the index exposed to the end user. It supports search functionality by using the functionality of the Content Router, and supports item insertion and deletion by using the functionality of the Data Store.

3. P-RING DATA STORE

The main challenge in devising a Data Store for P2P range indices is handling data skew. We would like the items to be uniformly distributed among the peers so that the load is nearly evenly distributed among the peers. Most existing P2P indices achieve this goal by hashing. Items are assigned to peers based on the hash value of their search key. Such an assignment has been shown to be very close to a uniform distribution with high probability [23]. However, hashing destroys the value ordering among the search key values, and thus cannot be used to process range queries efficiently (for the same reason that hash indices cannot be used to handle range queries efficiently).

For P-Ring to support range queries, we assign items to peers directly based on their search key value. In this case, the ring ordering is the same as the search key value ordering, wrapped around the highest value. The problem is that now, even in a stable P2P system with no peers joining or leaving, some peers might become overloaded due to skewed data insertions and/or deletions. We need a way to dynamically reassign and maintain the ranges associated to the peers. This section presents our algorithms for handling data skew. All our algorithms guarantee correctness in face of concurrent operations, as we can apply the techniques introduced by Linga et al. [15].

3.1 Handling Data Skew

The search key space is ordered on a ring, wrapping around the highest value. The Data Store partitions this ring space into ranges and assigns each of these ranges to a different peer. Let $p.range = (p.lb, p.ub]$ denote the range assigned to p. All items in the system with search key in $p.range$ are said to be *owned* by p. Let $p.own$ denote the list of all these items. Let $|p.own|$ denote the number of items in $p.own$ and hence in $p.range$. The number of ranges is less than the total number of peers in the system and hence there are some peers which are not assigned any range. Such peers are called *helper peers*. The others are called *owner peers*. Let $\mathcal{P}$ denote the set of all peers, and let $\mathcal{O}$ be the subset of owner peers in $\mathcal{P}$. Using these notations, the load imbalance is defined as $\frac{max_{p \in \mathcal{O}} |p.own|}{min_{p \in \mathcal{O}} |p.own|}$. In this section, we present algorithms to maintain the load imbalance at not more than two.

Analogous to B+-tree leaf page maintenance, the number of items in every range is maintained between bounds $\ell = \texttt{sf}$ and $u = 2 \cdot \texttt{sf}$, where sf is the "storage factor", a parameter we will talk more about in Section 3.2. Whenever the number of items in p's Data Store becomes larger than u (due to many insertions into $p.range$), we say that an *overflow* occurred. In this case, p tries to *split* its assigned range (and implicitly its items) with a helper peer. Whenever the number of items in p's Data Store becomes smaller than $\ell = \texttt{sf}$ (due to deletions from $p.range$), we say that an *underflow* occurred. Peer p tries to acquire a larger range and more items from its successor in the ring. In this case, the successor either *redistributes* its items with p, or gives up its entire range to p and becomes a helper peer.

Example Consider the Data Store in Figure 3 which shows the helper peers p_6 and p_7, and the ranges and search key values of items assigned to the other peers in the system (range $(5, 10]$ with items with search keys 6 and 8 are assigned to peer p_1 etc.). Assume that sf is 1, so each peer in the ring can have 1 or 2 items. When an item with search key 7 is inserted into the system, it will be stored at p_1, leading to an overflow. As shown in Figure 4, the range $(5, 10]$ is split between p_1 and the helper peer p_6. p_6 becomes the successor of p_1 on the ring and p_6 is assigned the range $(7, 10]$ with item with search key 8.

Split Algorithm 1 shows the pseudo-code of the `split` algorithm executed by a peer p that overflows. We use the notation $p::fn()$ when function $fn()$ is invoked at peer p, and $p.ringNode$ refers to the Fault Tolerant Ring component of the P-Ring at peer p. During a split, peer p tries to find a helper peer p' (see Section 3.2) and transfer half of its items, and the corresponding range, to p'. After p' is found (line 1), half of the items are removed from $p.own$ and $p.range$ is split accordingly. Peer p then invites peer p' to join the ring as its successor and maintain $p'.range$. The main steps of the algorithm executed by the helper peer p' are shown in Algorithm 2. Using the information received from p, p' initializes its Data Store component, the Ring component and the other index components above the Data Store.

Merge and Redistribution If there is an underflow at peer p, p executes the `merge` algorithm given in Algorithm 3. Peer p invokes the `initiateMergeMsgHandler` function on its successor on the ring. The successor sends back the action decided, `merge` or `redistribute`, a new range $newRange$ and the list of items $newItemsList$ that are to be re-assigned to p (line 2). p appends $newRange$ to $p.range$ and $newItemsList$ to $p.own$.

The outline of the `initiateMergeMsgHandler` function is given in Algorithm 4. The invoked peer, $p' = \texttt{succ}(p)$, checks whether a redistribution of items is possible between the two "siblings" (line 1). If yes, it sends some of its items and the corresponding range to p. If a redistribution is not possible, p' gives up all its items and its range to p, and becomes a helper peer.

Algorithm 1 : p.`split()`

```
1: p' = getHelperPeer();
2: if p' == null then
3:     return;
4: end if
5: //execute the split
6: splitItems = p.own.splitSecondHalf();
7: splitValue = p.own.lastValue();
8: splitRange = p.range.splitLast(splitValue);
9: p'::joinRingMsgHandler(p,splitItems,splitRange);
```

Algorithm 2 : p'.`joinRingMsgHandler`(p, $splitItems$, $splitRange$)

```
1: p'.range = splitRange;
2: p'.own = splitItems;
3: p'.ringNode.joinRing(p);
```

Algorithm 3 : p.`merge()`

```
1: //send message to successor and wait for result
2: (action, newRange, newItemsList) =
   p.ringNode.getSuccessor()::
   initiateMergeMsgHandler(p, |p.own|);
3: p.own.add(newItemsList);
4: p.range.add(newRange);
```

Algorithm 4 : $(action, newRange, newItemsList)$ p'.`initiateMergeMsgHandler`(p,$numItems$)

```
1: if numItems + |p'.own| > 2 · sf then
2:     //redistribute
3:     compute nbItemsToGive;
4:     splitItems = p'.own.splitFirst(nbItemsToGive);
5:     splitValue = splitItems.lastValue();
6:     splitRange = p'.range.splitFirst(splitValue);
7:     return (redistribute,splitRange,splitItems);
8: else
9:     //merge and leave the ring
10:    splitItems = p'.own;
11:    splitRange = p'.range;
12:    p'.ringNode.leaveRing();
13:    return (merge, splitRange, splitItems);
14: end if
```

Example. Consider again Figure 3 and assume that item with search key value 19 is deleted. Now there is an underflow at peer p_4 and peer p_4 calls `initiateMergeMsgHandler` in p_5. Since p_5 has only one item, redistribution is not possible. Peer p_5 sends its item to p_4 and becomes a helper peer, with no range to own. As shown in Figure 5, peer p_4 now owns the whole range $(18, 5]$.

3.2 Managing Helper Peers

We first discuss the pros and cons of using helper peers.

The main advantage of using helper peers is the decrease in cost of re-balancing operations. Ganesan et al. showed in [11] that any efficient load-balancing algorithm that guarantees a constant imbalance ratio, as our algorithm does, needs to use re-order operations. A highly loaded peer finds a lightly loaded peer that can give its load to some neighbor peer, and take over some of the load of the

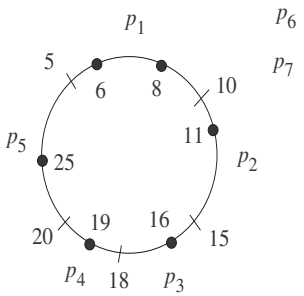

Figure 3. Data Store (DS)

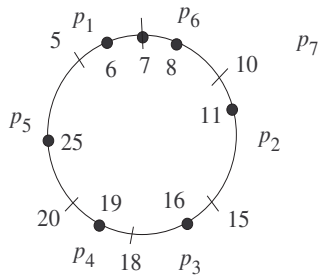

Figure 4. DS After Split

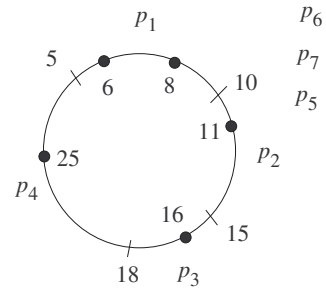

Figure 5. DS After Merge

highly loaded peer. In all of the previous approaches to load balancing that we are aware of [11, 14, 13], the lightly loaded peer is already part of the ring, so it needs to leave the ring before joining it in a new place. Leaving the ring is an expensive operation: new neighbors are established in the ring, the items of the peer are sent to the neighbor(s), more replicas are created to compensate for the loss of the replicas stored at the leaving peer, and finally, the routing structure adjusts for the change. By using helper peers that are not part of the ring, all these costs are eliminated, leading to more efficient and faster load balancing.

Using helper peers might seem to contradict the symmetry of the P2P systems. However, the number of helper peers is usually small, and they change over time, due to re-balancing. Moreover, Section 3.3 introduces a scheme that uses the helper peers for load balancing among all peers.

Now, let us see how we manage the helper peers. Recall that helper peers are used during split and are generated during merge. There are three important issues to be addressed. First, we need a reliable way of "storing" and finding helper peers. Second, we need to ensure that a helper peer exists when it is needed during split. Finally, even though helper peers do not have a position on the ring, they should be able to query the data in the system.

To solve the first issue, we create an artificial item $(\perp, p'.address)$ for every helper peer p', where $\perp$ is the smallest possible search key value. This item is inserted into the system like any regular item. When a helper peer is needed, an equality search for $\perp$ is issued. The search is processed as a regular query and the result is returned to the requester. As there are much fewer helper peers than regular items, managing helper peers does not significantly increase the load on the peers storing them. Using the HR router from Section 4, the cost of inserting, deleting, or finding a helper peer is $O(log_d P)$.

To ensure that a helper peer exists when an overflow occurs, we set `sf` = $\lceil \frac{N}{P} \rceil$. The number of items N and the number of peers P can be dynamically estimated by each peer at no additional message cost (see Section 4.3).

Finally, each helper peer maintains a list of owner peers to which it can forward any query to be processed.

3.3 Load Balancing using Helper Peers

The load-balancing scheme proposed maintains the number of items stored by each owner peer within strict bounds. However, the helper peers do not store any items, so there is no "true" load balance among peers. We propose now an extension to the basic scheme, which uses the helper peers to"truly" balance the load. The extended scheme is provably efficient, i.e., every insert and delete of an item has an amortized constant cost. Also the *load imbalance* is bounded by a small constant.

Observe that if we somehow assign items to helper peers too while maintaining the bounds ℓ and u on the number of items assigned, we are able to bound the load imbalance by $\frac{u}{\ell}$. We therefore extend the functionality of helper peers. Every helper peer is obliged to "help" an owner peer already on the ring. A helper peer helps an owner peer by managing some part of the owner peers range and hence some of its load. If owner peer p has k helpers $q_1, q_2, \ldots, q_k$, $p.range = (lb, ub]$ is divided into $(lb = b_0, b_1], (b_1, b_2], \ldots, (b_k, ub]$ such that each sub-range has equal number of items. Peer p is now *responsible* for $(b_k, ub]$. Each of p's helpers, q_j, becomes *responsible* for one of the other ranges, say $(b_{j-1}, b_j]$. Let $q.resp$ be the list of items peer q is *responsible* for and $q.range_{resp}$ be the corresponding range. q participates in routing and all queries dealing with $q.range_{resp}$ will reach q. However, p still owns all the items in $(lb, ub]$ and is responsible for initiating the load balance operations. Also, any insert or delete that reaches a helper peer is forwarded to the owner peer, who will ensure that the items own are evenly divided among itself and the helpers. In this context, the definition of the *load imbalance* becomes $\frac{\max_{p \in \mathcal{P}} |p.resp|}{\min_{p \in \mathcal{P}} |p.resp|}$. In this section, we provide algorithms to maintain the load imbalance at not more than $2 + \epsilon$, for any $\epsilon > 0$.

The extended scheme, EXTLOADBALANCE, is very similar to the basic scheme. In the split algorithm Algorithm 1, only step 1 changes, and a new step is added. If p already has helper peers, then p uses one of the helpers q to perform the split. The helper peer q will become an owner peer. Else, p issues a search for a helper peer. During the split, not only the range and items of p are split, but also the helper peers of p. At the end of the split, p redistributes the new $p.own$ with its new reduced set of helpers. In the merge algorithm, Algorithm 3, p contacts the first successor on the ring who is an owner peer, p'. If the decision is to merge, all the data items are moved from p to p' and p and all the associated helpers leave the ring and start helping some other randomly chosen peers. If the decision is to redistribute, the items are moved from $p'.own$ to $p.own$. The items in $p'.own$ and respectively $p.own$ are re-distributed among the helper peers of p' and respectively p.

To bound the load imbalance close to $\frac{u}{\ell}$, EXTLOADBALANCE has an additional load balancing operation called *usurp*. Algorithm 5 shows the pseudo-code of the `usurp` algorithm executed by an owner peer p. During *usurp*, an owner peer p can *usurp*, or take over, a helper peer q of another owner peer p', if $|p.resp| \geq 2\sqrt{1+\delta}|q.resp|$, for a given constant $\delta > 0$. The helper peer q starts helping p, so $|p.resp|$ is reduced. The `getLeastLoaded-HelperPeer()` function can be implemented by using the HR (Section 4.1) to maintain information about the least loaded peer.

Our algorithms will bound the load imbalance in the system by a small constant (2+ϵ), at an amortized constant cost for item insertions and deletions. We can prove the following:

THEOREM 1. *Consider an initial load balanced system. For every sequence σ of item inserts and deletes, and constants ℓ, u, ϵ*

Algorithm 5 : p.usurp()

```
//find least loaded helper peer and its "master"
(q,p') = getLeastLoadedHelperPeer();
if |p.resp| ≥ 2√(1+δ)|q.resp| then
    p.setHelperPeer(q);
    redistribute p.own among new set of helpers;
    redistribute p'.own among new set of helpers;
end if
```

such that $\frac{u}{\ell} \geq 2$ *and* $\epsilon > 0$*,*

- Load Imbalance*: The sequence of* split*,* merge *and* usurp *operations performed by Algorithm* EXTLOADBALANCE *is such that after the completion of any prefix of* σ*, the current partition of ranges, the assignment of ranges to owner peers and assignment of helper peers to owner peers satisfy*
  1. $\ell \leq |p.own| \leq u$ *for all owner peers* $p \in \mathcal{O}$*;*
  2. *load imbalance* $= \frac{\max_{q\in\mathcal{P}} |q.resp|}{\min_{q\in\mathcal{P}} |q.resp|} < \max(2+\epsilon, \frac{u}{\ell})$.
- Efficiency*: If* $\frac{u}{\ell} \geq 2+\epsilon$*, the above sequence of* split*,* merge *and* usurp *operations is such that the amortized cost of an insert or a delete operation is a constant.*

PROOF. Full proofs for all theorems are given in [7].

We sketch now the proof for bound on the load imbalance. Since $u \geq 2\ell$, it is easy to see that during the course of the algorithm EXTLOADBALANCE, the $split$ and the $merge$ operations bound the size of $p.own$ within ℓ and u. Unlike $p.own$, we can bound $\ell \leq |p.resp| \leq u$, only for all $p \in \mathcal{O}$ with no helper peers. An owner peer p with $|p.own| = \ell$ could have helpers making $|p.resp| < \ell$. However, thanks to $usurp$ operations, there cannot exists a $q \in \mathcal{P}$ such that $|q.resp| > 2(\sqrt{1+\delta}) \cdot |p.resp|$. By setting $\delta = (1+\epsilon/2)^2 - 1$, we get the required bound. □

We now sketch the proof for the efficiency result. Our cost model is very similar to the one used in Ganesan et al. [11]. The only cost we consider is the cost of moving items due to the load balancing operations. There are three major components to the cost: a) Data movement: we model this cost as being linear in the number of items moved from one peer to the other. b) Distributing items amongst helper peers, whenever the set of items owned by a peer p or the set of helpers $\mathcal{H}(p)$ changes: we use $|p.own|$ as a very conservative estimate of the number of items moved in this case. c) Load information: our algorithm requires non-local information about the least loaded helper peer. We assume that this cost can be included in the data movement cost.

Let $\ell = \mathtt{sf}$ and $u = (2+\epsilon)\mathtt{sf}$ for some $\epsilon > 0$. Recall that $\mathtt{sf} = \lceil \mathtt{N}/\mathtt{P} \rceil$. To prove the amortized constant cost of insert and delete we use the potential function $\Phi = \Phi_o + \Phi_r$, where Φ_o is the *ownership potential* and Φ_r is the *responsibility potential* defined as follows:

$\Phi_o = \sum_{p\in\mathcal{P}} \phi_o(p)$, where

$$\phi_o(p) = \begin{cases} 0 & p \notin \mathcal{O} \text{ (helper peer)} \\ \frac{c_o}{\mathtt{sf}}(l_0 - |p.own|)^2 & \mathtt{sf} \leq |p.own| \leq l_0 \\ 0 & l_0 \leq |p.own| \leq u_0 \\ \frac{c_o}{\mathtt{sf}}(|p.own| - u_0)^2 & u_0 \leq |p.own| \leq (2+\epsilon)\mathtt{sf} \end{cases}$$

$$l_0 = (1+\frac{\epsilon}{4})\mathtt{sf}$$

$$u_0 = (2+\frac{3\epsilon}{4})\mathtt{sf}$$

$\Phi_r = \sum_{q\in\mathcal{P}} \phi_r(q)$, where $\phi_r(q) = \frac{c_r}{\mathtt{sf}}(|q.resp|)^2$, and constants c_o and c_r will be defined later.

We show that the increase in the potential Φ due to an insert or delete is bounded by a constant, and the (maximum) cost of a re-balancing operation is smaller than the (minimum) decrease in the potential Φ due to re-balancing. These facts prove that the amortized cost of an insert or delete operation is constant.

Insert: During insert operation, an item is inserted into $p.own$ for some p and inserted into $q.resp$ for some $q \in \mathcal{H}(p) \cup \{p\}$. $\phi_r(q)$ increases, while $\phi_o(p)$ increases if $u_0 \leq |p.own| \leq (2+\epsilon)\mathtt{sf}$, and decreases if $\mathtt{sf} \leq |p.own| \leq l_0$. The maximum increase in Φ occurs when both $\phi_r(q)$ and $\phi_o(p)$ increase and this increase is

$$\begin{aligned}
\Delta_{ins}\Phi_o &= \frac{c_o}{\mathtt{sf}}(|p.own|+1-u_0)^2 - \frac{c_o}{\mathtt{sf}}(|p.own|-u_0)^2 \\
&= \frac{c_o}{\mathtt{sf}}(2|p.own|+1-2u_0) \\
&\leq \frac{c_o}{\mathtt{sf}}(2(2+\epsilon)\mathtt{sf}+1-2(2+\frac{3\epsilon}{4})\mathtt{sf}) \\
&\leq \frac{c_o}{\mathtt{sf}}(\frac{\epsilon}{2}\mathtt{sf}+1) \leq \frac{\mathtt{c_0}\epsilon}{2} + \mathtt{c_o} \\
\Delta_{ins}\Phi_r &= \frac{c_r}{\mathtt{sf}}(|q.resp|+1)^2 - \frac{c_r}{\mathtt{sf}}(|q.resp|)^2 \\
&= \frac{c_r}{\mathtt{sf}}(2|q.resp|+1) \\
&\leq \frac{c_r}{\mathtt{sf}}(2(2+\epsilon)\mathtt{sf}+1) \leq 2(2+\epsilon)c_r + c_r
\end{aligned}$$

$$\Delta_{ins}\Phi \leq \frac{c_o\epsilon}{2} + 2(2+\epsilon)c_r + c_o + c_r \qquad (1)$$

Delete: During delete operation, an item is deleted from $p.own$ for some peer p and deleted from $q.resp$ for some $q \in \mathcal{H}(p) \cup \{p\}$. Analogous to the insert case, we can show that the maximum increase in potential to be

$$\Delta_{del}\Phi \leq \frac{c_o\epsilon}{2} + 2(2+\epsilon)c_r + c_o + c_r \qquad (2)$$

We showed in 1 and 2 that the increase in the potential Φ due to an insert or delete is bounded by a constant. We show now that the maximum cost of a re-balancing operation (split, merge or re-distribute, and usurp) is smaller than the minimum decrease in the potential Φ due to that re-balancing operation.

Split: First let us look at the decrease in the ownership potential $\Delta_{split}\Phi_o$. During a split, a peer p owning $|p.own| = (2+\epsilon)\mathtt{sf}$ items, gives half of its items to a helper peer q. After the split, both p and q own $(1+\frac{\epsilon}{2})\mathtt{sf}$ items. Hence, the final ownership potentials of p and q are 0. Also, the initial ownership potential of q is 0 since before the split q was not an owner peer.

$$\Delta_{split}\Phi_o \geq \frac{c_o}{\mathtt{sf}}((2+\epsilon)\mathtt{sf} - \mathtt{u_0})^2 = \mathtt{c_o}(\frac{\epsilon}{4})^2\mathtt{sf}$$

Next, consider the change in the responsibility potential. When $\mathcal{H}(p) \neq \emptyset$, q is chosen from $\mathcal{H}(p)$ and the helper peers are distributed amongst p and q evenly. In this case, the responsibilities change only when the number of helpers apart from q (i.e., $|\mathcal{H}(p) \setminus \{q\}|$) is odd, say $2h+3$. This is because the $(1+\frac{\epsilon}{2})\mathtt{sf}$ items in p and q are distributed amongst $h+1$ and $h+2$ peers

respectively. In this case the decrease in Φ_r would be

$$\begin{aligned}
\Delta_{split}\Phi_r &= (2h+3)\frac{c_r}{\mathtt{sf}}\left(\frac{(2+\epsilon)\mathtt{sf}}{2h+3}\right)^2 \\
&\quad -(h+1)\frac{c_r}{\mathtt{sf}}\left(\frac{\left(1+\frac{\epsilon}{2}\right)\mathtt{sf}}{h+1}\right)^2 \\
&\quad -(h+2)\frac{c_r}{\mathtt{sf}}\left(\frac{\left(1+\frac{\epsilon}{2}\right)\mathtt{sf}}{h+2}\right)^2 \\
&\geq \frac{c_r}{\mathtt{sf}}\left(\frac{((2+\epsilon)\mathtt{sf})^2}{4(h+2)} - \frac{((2+\epsilon)\mathtt{sf})^2}{4(h+1)}\right) \\
&= c_r(2+\epsilon)^2\mathtt{sf}\left(\frac{1}{4(\mathtt{h}+2)} - \frac{1}{4(\mathtt{h}+1)}\right) \\
&= -\frac{c_r(2+\epsilon)^2}{4(h+1)(h+2)}\mathtt{sf} \\
\Delta_{split}\Phi_r &\geq -\frac{c_r(2+\epsilon)^2}{8}\mathtt{sf}
\end{aligned}$$

When $\mathcal{H}(p) = \emptyset$, p splits its items with a peer q, where $q \in \mathcal{H}(p_2)$ for some p_2. Let $h_2 = |\mathcal{H}(p_2)|$, $l_2 = |p_2.own|$. We have,

$$\begin{aligned}
\Delta_{split}\phi_r(p) &= \frac{c_r}{\mathtt{sf}}\left((2+\epsilon)\mathtt{sf})^2 - \left(\left(1+\frac{\epsilon}{2}\right)\mathtt{sf}\right)^2\right) \\
\Delta_{split}\phi_r(q) &= \frac{c_r}{\mathtt{sf}}\left(\left(\frac{l_2}{1+h_2}\right)^2 - \left(\left(1+\frac{\epsilon}{2}\right)\mathtt{sf}\right)^2\right)
\end{aligned}$$

$\forall q_2(\neq q) \in \mathcal{H}(p) \cup \{p_2\}$,

$$\begin{aligned}
\Delta_{split}\phi_r(q_2) &= \frac{c_r}{\mathtt{sf}}\left(\left(\frac{l_2}{1+h_2}\right)^2 - \left(\frac{l_2}{h_2}\right)^2\right) \\
\Delta_{split}\Phi_r &= \frac{c_r}{\mathtt{sf}}\left(\frac{1}{2}((2+\epsilon)\mathtt{sf})^2 + \frac{\mathtt{l}_2^2}{1+\mathtt{h}_2} - \frac{\mathtt{l}_2^2}{\mathtt{h}_2}\right) \\
&= \frac{c_r}{\mathtt{sf}}\left(\frac{((2+\epsilon)\mathtt{sf})^2}{2} - \frac{l_2^2}{h_2(h_2+1)}\right) \\
&\geq \frac{c_r}{\mathtt{sf}}\left(\frac{((2+\epsilon)\mathtt{sf})^2}{2} - \frac{((2+\epsilon)\mathtt{sf})^2}{1\cdot 2}\right) \\
&\geq 0
\end{aligned}$$

Hence the minimum decrease in the potential due to a split is $\Delta_{split}\Phi \geq c_o(\frac{\epsilon}{4})^2\mathtt{sf}$.

Now, we show that the cost of a split operation is at most $(3+\frac{3\epsilon}{2})\mathtt{sf}$. In the case when p, the splitting peer, has a helper q, the cost is contributed by the transfer of $(1+\frac{\epsilon}{2})\mathtt{sf}$ items and the re-distribution of items amongst p's and q's helpers. When p does not have any helpers, p splits with the helper q of some other owner peer p_2. Here the cost involves transfer of items from p to q and the re-distribution of items amongst p_2's remaining helpers.

In order to have the cost of split lower than the decrease in the potential due to split, we need c_o and c_r such that

$$c_o\left(\frac{\epsilon}{4}\right)^2\mathtt{sf} \quad \geq \quad \left(3+\frac{3\epsilon}{2}\right)\mathtt{sf} \tag{3}$$

Redistribute: Analogous to the change in potential due to split, we can prove that the minimum decrease in potential due to redistribute is $\Delta_{redist}\Phi \geq c_o\mathtt{sf}\frac{\epsilon^2}{16} - \mathtt{c_r}\mathtt{sf}\left(\frac{\epsilon}{2}+\frac{\epsilon^2}{8}\right)$.

The cost of the redistribute operation, is at most $(3+\frac{5\epsilon}{4})\mathtt{sf}$. The cost involves transfer of $\frac{\epsilon}{4}\mathtt{sf}$ items and the redistribution of the final set of items owned by the two peers involved in the transfer amongst their helpers.

The cost-potential equation for a redistribute operation becomes:

$$c_o\mathtt{sf}\frac{\epsilon^2}{16} - \mathtt{c_r}\mathtt{sf}\left(\frac{\epsilon}{2}+\frac{\epsilon^2}{8}\right) \quad \geq \quad \left(3+\frac{5\epsilon}{4}\right)\mathtt{sf} \tag{4}$$

Merge: Again analogous to the split case, we can prove that the minimum decrease in potential due to merge is $\Delta_{merge}\Phi \geq c_o\left(\frac{\epsilon}{4}\right)^2\mathtt{sf} - 2\mathtt{c_r}\left(1+\frac{\epsilon}{4}\right)\mathtt{sf}$

The cost of a merge is at most $(3+\frac{\epsilon}{2})\mathtt{sf}$, since the cost only involves transfer of $\mathtt{sf}$ items to the more loaded peer and redistribution of at most $(2+\frac{\epsilon}{2})\mathtt{sf}$ items amongst the new set of helper peers.

The cost-potential equation for a merge operation becomes:

$$c_o\left(\frac{\epsilon}{4}\right)^2\mathtt{sf} - 2\mathtt{c_r}\left(1+\frac{\epsilon}{4}\right)\mathtt{sf} \quad \geq \quad \left(3+\frac{\epsilon}{2}\right)\mathtt{sf} \tag{5}$$

Usurp: We can prove that the minimum decrease in potential due to an usurp operation is $\Delta_{usurp}\Phi \geq \frac{2c_r\delta}{\kappa_h^2}\mathtt{sf}$, where $\kappa_h = (4+\epsilon)(4(2+\epsilon)\sqrt{1+\delta}-1)$ (κ_h is the maximum number of helper peers assigned to an owner peer).

The usurp operation costs $\ell_1 + \ell_2 \leq 2(2+\epsilon)\mathtt{sf}$, where the two non free peers involved own ℓ_1 and ℓ_2 items respectively. The cost arises due to the redistribution amongst the new set of helpers. The cost-potential equation for an usurp operation becomes:

$$\frac{2c_r\delta}{\kappa_h^2}\mathtt{sf} \quad \geq \quad 2(2+\epsilon)\mathtt{sf} \tag{6}$$

Solving equations 3, 4, 5, 6, we get

$$\begin{aligned}
c_o\frac{\epsilon^2}{16} &\geq c_r\left(1+\epsilon+\frac{\epsilon}{4}\right) + \left(3+\frac{3\epsilon}{2}\right) \\
c_r &\geq \frac{(2+\epsilon)\kappa_h^2}{\delta}
\end{aligned}$$

By setting the constants c_r and c_o to values as shown above, we can prove that the amortized cost of inserts and deletes is a constant when $\mathtt{sf}$ does not change. The proof for the amortized constant cost for insert/delete in the case where $\mathtt{sf}$ does change due to the change in the number of items in the system, is omitted here due to space constraints. □

4. P-RING CONTENT ROUTER

The goal of our Content Router is to efficiently route messages to peers in a given range. The main challenge is to handle skewed distributions. Since the search keys can be skewed, the peer ranges may not be of equal length.

We devise a new Content Router called *Hierarchical Ring* (or short, HR) that can handle highly skewed distributions. In this section we describe the content router, the routing algorithm and the maintenance algorithms. We then give analytical bounds for the search performance in a stable system and under heavily skewed insertion patterns.

4.1 Hierarchical Ring

The HR is based on the simple idea of constructing a hierarchy of rings. Let d be an integer > 1, called the *order* of HR. At the lowest level, level 1, each peer p maintains a list of the first d successors on the ring. Using the successors, a message could always be forwarded to the last successor in the list that does not overshoot the target, "skipping" up to d-1 peers at a time. For instance, Figure 6 shows a hierarchy of rings with order (d) 2. As shown, peer p_1 is responsible for the range $(5, 10]$, p_2 is responsible for $(10, 15]$ and so on. Each peer knows its successor on the ring: $\mathtt{succ}(p_1) = p_2$, $\mathtt{succ}(p_2) = p_3$, and so on. At level 1 in the

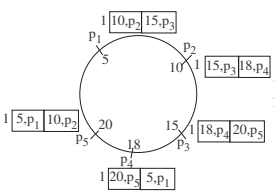

Figure 6. HR Level 1

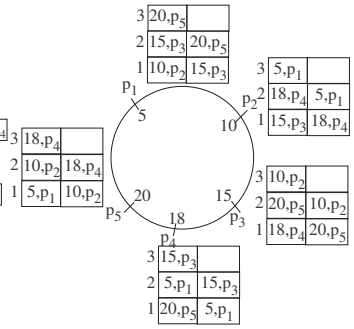

Figure 7. HR Levels 1, 2, and 3

HR, each peer maintains a list of 2 successors, as shown. Suppose p_1 needs to route a message to a peer with value 20. p_1 will route the message to p_3 and p_3 will forward the message to p_5, the final destination.

At level 2, we again maintain a list of d successors. However, a successor at level 2 corresponds to the dth successor at level 1. Using these successors, a message can always be routed to the last successor in the list that does not overshoot the target, "skipping" up to $d^2 - 1$ peers at a time. Figure 7 shows the content of level 2 nodes at each peer in the ring. If p_1 needs to route a message to a peer with value 20, p_1 will route the message directly to p_5 (the final destination), using the list at level 2. The procedure of defining the successor at level $l+1$ and creating a list of level $l+1$ successors is iterated until no more levels can be created. In Figure 7, for peer p_1, $\texttt{succ}_3(p_5) = p_4$, which overshoots p_1, so no more levels can be constructed for p_1.

Note that we are conceptually indexing $positions$ in the ring (i.e. at level l, a peer p has pointers to peers that are d^l $peers$ away) instead of values, which allows HR to perform well, regardless of the data distribution.

Formally, the data structure for a HR of order d is a doubly indexed array $node[level][position]$, where $1 \leq level \leq numLevels$ and $1 \leq position \leq d$. The HR is defined to be consistent if and only if at each peer p:

- $p.node[1][1] = \texttt{succ}(p)$
- $p.node[1][j+1] = \texttt{succ}(p.node[1][j])$, $1 \leq j < d$
- $p.node[l+1][1] = p.node[l][d]$,
- $p.node[l+1][j+1] = p.node[l+1][j].node[l+1][1]$, $1 \leq l < numLevels$, $1 \leq j < d$
- The successor at $numLevels$ of the last peer in the list at $numLevels$ level "wraps" around, so all the peers are indeed indexed: $p.node[numLevels].lastPeer.node[numLevels][1] \in [p, p.node[numLevels].lastPeer)$

From this definition, it is easy to see that a consistent HR of order d, has at most $\lceil \log_d P \rceil$ levels, and the space requirement for the HR at each peer is $O(d \cdot \log_d P)$.

4.2 Maintenance

Peer failures and insertions, as well as splits and merges at the Data Store level (perceived as peer insertions, and respectively departures, at the Content Router level), disrupt the consistency of the HR. We have a remarkably simple *Stabilization Process* that runs periodically at each peer and repairs the inconsistencies in the HR. The algorithm guarantees that the HR structure eventually becomes fully consistent after any pattern of concurrent insertions and deletions, as long as the peers remain connected at the ring level.

Algorithm 6 : p.Stabilize()

```
1: i = 1;
2: repeat
3:     root=p.StabilizeLevel(i);
4:     i++;
5: until (root)
6: p.numLevels = i − 1;
```

Algorithm 7 : p.StabilizeLevel(int i)

```
1: succEntry = p.node[i][1];
2: p.node[i] = succEntry.node[i];
3: INSERT(i, succEntry);
4: if p.node[i].lastPeer.node[i][1] ∈
   [p, p.node[i].lastPeer) then
5:     return true
6: else
7:     p.node[i+1][1] = p.node[i][d];
8:     return false;
9: end if
```

The stabilization process is important for the performance of the queries, but not for their correctness. As long as the peers are connected at the ring level, queries can be processed by forwarding them along the successor pointers. The stabilization process fixes the inconsistencies in the HR in order to provide logarithmic search performance for queries. We chose to have a periodic stabilization process that repairs the inconsistencies in the HR over performing reactive repairs, as the latter can lead to high maintenance costs in case of high churn [20]. Using a periodic stabilization mechanism is similar to most other P2P index structures [4, 19, 21, 23].

The algorithm executed periodically by the Stabilization Process is shown in Algorithm 6. The algorithm loops from the lowest level to the top-most level of the HR until the highest (root) level is reached (as indicated by the boolean variable *root*). Since the height of the HR data structure could actually change, we update the height ($p.numLevels$) at the end of the function.

Algorithm 7 describes the Stabilization Process within each level of the HR structure at a peer. The key observation is that each peer needs only local information to compute its own successor at each level. Thus, each peer relies on other peers to repair their own successor at each level. When a peer p stabilizes a level, it contacts its successor at that level and asks for its entries at the corresponding level. Peer p replaces its own entries with the received entries and inserts its successor as the first entry in the index node (lines 2 and 3). The INSERT procedure inserts the specified entry at the beginning of the list at given level, and it ensures that no more than d entries are in the list and none of the entries in the list overshoots p (if the list does wraps around, this should be the last level). Line 4 checks whether this level should be the last level in the HR. This is the case if all the peers in the system are already covered. If this level is not the root level, the stabilization procedure computes the successor at the higher level (line 7) and returns.

The periodic Stabilization Process runs independently at each peer, without the need for synchronization. Regardless of the order in which the Stabilization Process is run at different peers (stabilization of some level i in the HR structure at some peers might occur before the stabilization at level $i-1$ at some other peers), the Stabilization Process will move the HR structure towards a more consistent state. Eventually, the entire HR becomes consistent, as shown in Theorem 2.

Definition We define a stabilization unit (`su`) to be the time needed

to run the `StabilizeLevel` procedure at some level in all peers.

THEOREM 2 (STABILIZATION TIME). *Given that at time t there are P peers in the system, the fault tolerant ring is connected and the successor pointers are correct, and the stabilization procedure starts running periodically at each peer, at time $t + (d-1)\lceil \log_d P \rceil$su the HR is consistent with respect to the P peers, if no peers fail.*

Due to space constraints, we omit the proof here; full proofs for all theorems are given in the technical report [7].

4.3 Storage Factor Estimation

The algorithms in Section 3 have one parameter: the storage factor `sf` or ℓ, the required minimum number of items stored by a peer. `sf` depends on $\frac{N}{P}$ (N = number of items and P = number of peers). Each peer estimates N and P as follows. Each entry p' in the HR at a peer p stores two additional counters to estimate the number of peers and the number of items in the range $(p, p']$. These counters are aggregated bottom-up and the highest-level values are used as estimates for N and P. Maintaining the counters does not increase the number of messages in the system, as we piggyback the numbers on the HR stabilization messages. Our experiments show that P-Ring achieves a load imbalance of approximately two, even in a dynamic system, which proves that the estimated `sf` is accurate.

4.4 Routing

The Content Router component supports the `sendReceive(msg,range)` primitive. We assume that each routing request originates at some peer p in the P2P system. For simplicity of presentation, we assume that the `range` has the form $(lb, ub]$.

The routing procedure shown in Algorithm 8 takes as input the lower-bound (lb) and the upper-bound (ub) of the requested range, the message to be routed, and the address of the peer where the request originated. `rangeMin(p)` denotes the low end value of $p.range$, and $p.node[i][j].iValue$ and $p.node[i][j].peer$ denote the value, and respectively the address of the peer stored in the HR entry $p.node[i][j]$ (we used $p.node[i][j].iValue$ = `rangeMin(p.node[i][j].peer)`). Each peer selects the farthest away pointer that does not overshoot lb and forwards the request to that peer. Once the algorithm reaches the lowest level of the HR, it traverses the successor list until the value of a peer exceeds ub (lines 8-9). Note that every peer which is responsible for a part of $(lb, ub]$ is visited during the traversal along the ring. At the end of the range scan, a $RoutingDoneMessage$ is sent to the originator (line 11).

Example: Consider a routing request for the range $(18, 25]$ that is issued at peer p_1 in Figure 7. The routing algorithm first determines the highest HR level in p_1 that contains an entry whose value is between 5 (value of p_1) and 18 (the lower bound of the range query). In the current example, this corresponds to the first entry at the second level of p_1's HR nodes, which points to peer p_3 with value 15. The routing request is hence forwarded to p_3. p_3 follows a similar protocol, and forwards the request to p_4 (which appears as the first entry in the first level in p_3's HR nodes). Since p_4 is responsible for items that fall within the required range, p_4 processes the routed message and returns the results to the originator p_1 (line 6). Since the successor of p_4, p_5, might store items in the $(18, 25]$ range, the request is also forwarded to p_5. p_5 processes the request and sends the results to p_1. The search terminates at p_5 as the value of its successor (5) does not fall within the query range.

Algorithm 8 : p.`routeHandler`(lb, up, msg, $originator$)

```
1:  // find maximum level that contains an
2:  // entry that does not overshoot lb.
3:  find the maximum level l such that ∃ j > 0
    such that p.node[l][j].iValue ∈ (rangeMin(p), lb].
4:  if no such level exists then
5:     //handle the message and send the reply
6:     send(p.handleMessage(msg), originator);
7:     if rangeMin(succ(p)) ∈ (rangeMin(p), ub] then
8:        // if successor satisfies search criterion
9:        send(Route(lb,ub,msg,originator,requestType),
          succ(p));
10:    else
11:       send(RoutingDoneMessage,originator);
12:    end if
13: else
14:    find maximum k such that
       p.node[l][k].iValue ∈ (rangeMin(p), lb];
15:    send(Route((lb,ub,msg,originator),
       p.node[l][k].peer));
16: end if
```

In a consistent state, the routing will go down one level in the HR every time a routing message is forwarded in line 15. This guarantees that we need at most $\lceil \log_d P \rceil$ steps to find lb, if the HR is consistent. If the HR is inconsistent, the routing cost may be more than $\lceil \log_d P \rceil$. Even if the HR is inconsistent, it can still route requests by using the entries to the maximum extent possible, and then sequentially scanning along the ring. In Section 6.4, we show that the search performance of HR does not degrade much even when the index is temporarily inconsistent.

It is important to note that in a P2P system we cannot guarantee that every route request terminates. For example, a peer p could crash in the middle of processing a request, in which case the originator of the request would have to time out and try the routing request again. This model is similar to that used in most other P2P systems. [19, 21, 23].

We can formally prove the following properties of routing in Hierarchical Ring.

THEOREM 3 (SEARCH PERFORMANCE IN STABLE STATE). *In a stable system of P peers with a consistent HR structure of order d, range queries take at most $\lceil \log_d P \rceil + m$ hops, where m is the number of peers in the requested range.*

PROOF. From the definition of HR data structure, a consistent HR of order d for P peers has $\lceil \log_d P \rceil$ levels. In a consistent HR, the routing procedure goes down one level every time a routing message is forwarded to another peer. The peer with the lowest value in the requested range is found once the lowest level is reached. After the first answer is found, all the other answers are found by following the successor links. This ensures that the maximum number of hops needed to answer a range query in a stable HR is $\lceil \log_d P \rceil + m$, where m is the number of peers in the requested range. □

THEOREM 4 (SEARCH PERFORMANCE DURING INSERTIONS). *If we have a stable system with a consistent HR of order d and we start inserting peers at the rate r peers/stabilization unit, range queries take at most $\lceil \log_d P \rceil + 2r(d-1)\lceil \log_d P \rceil + m$ hops, where P is the current number of peers in the system, and m is the number of peers in the requested range.*

PROOF. Let t_0 be the initial time and P_0 be the number of peers in the system at time t_0. For every $i > 0$ we define t_i to be $t_{i-1} + (d-1)\lceil \log_d(P_{i-1})\rceil \cdot \mathtt{su}$ and P_i to be the number of peers in the system at time t_i. We call an *old* peer to be a peer that can be reached in at most $\lceil \log_d P\rceil$ hops using the HR. If a peer is not *old*, we call it *new*. At any time point, the worst case search cost for equality queries is $\lceil \log_d P\rceil + x$, where $\lceil \log_d P\rceil$ is the maximum number of hops using the HR to find an old peer and x is the number of new peers. x is also the maximum number of hops to be executed using the successor pointers to find any one of the new x peers (the worst case is when all new peers are successors in the ring).

We show by induction on time that the number of new peers in the system at any time is at most $2r(d-1)\lceil \log_d P\rceil$, which proves the theorem.

As the base induction step we prove that at any time point in the interval $[t_0, t_1]$ there are no more than $2r(d-1)\lceil \log_d P\rceil$ new peers and at time t_1 there are no more than $rd\lceil \log_d P\rceil$ new peers. From hypothesis, at t_0 the HR is consistent, so there are no new peers. At the insertion rate of r peers/$\mathtt{su}$, at any time point in $[t_0, t_1]$, the maximum number of peers inserted is $r(d-1)\lceil \log_d(P_0)\rceil$, which is smaller than $r(d-1)\lceil \log_d P\rceil$. This proves both statements of the base induction step.

We prove now that if the maximum number of new peers at time t_i is $rd\lceil \log_d P\rceil$, than, at any time point in $[t_i, t_{i+1}]$ the maximum number of new peers is $2r(d-1)\lceil \log_d P\rceil$ and the maximum number of new peers at time t_{i+1} is $r(d-1)\lceil \log_d P\rceil$, where $i \geq 1$. The maximum number of peers inserted between t_i and t_{i+1} is $r(d-1)\lceil \log_d(P_i)\rceil$ which is smaller than $r(d-1)\lceil \log_d P\rceil$. From the induction hypothesis, at time t_i there were at most $r(d-1)\lceil \log_d P\rceil$ new peers. Between t_i and t_{i+1}, some old peers can become new and new peers can become old, due to changes in the HR structure. However, the total number of entries in the HR structure does not decrease, so the number of old peers becoming new cannot be higher than the number of new peers becoming old. Out of the peers in the system at time t_i, at most $r(d-1)\lceil \log_d P\rceil$ of them are new at any time between t_i and t_{i+1}. Adding the peers inserted since t_i we get that at any time point in $[t_i, t_{i+1}]$ the maximum number of new peers is $2r(d-1)\lceil \log_d P\rceil$. From Theorem 2, at time t_{i+1}, all the peers existing in the system at time t_i are integrated into the HR structure. This means that all peers existing at time t_i are/became old peers at time t_{i+1}, which leaves the maximum number of new peers at time t_{i+1} to be at most $r(d-1)\lceil \log_d P\rceil$ (the peers inserted between t_i and t_{i+1}).

From induction it follows that at any time, the maximum number of new peers is no more than $2r(d-1)\lceil \log_d P\rceil$, which means that equality queries take at most $\lceil \log_d P\rceil + 2r(d-1)\lceil \log_d P\rceil$ hops. □

5. RELATED WORK

Most of the indexing techniques developed for distributed databases (e.g., [16, 17, 18]) are not designed for highly dynamic peers and therefore are not appropriate for a P2P environment.

CAN [19], Chord [23], Pastry [21] and Tapestry [24] implement distributed hash tables to provide efficient lookup of a given key value. Since a hash function destroys the ordering in the key value space, these structures cannot process range queries efficiently.

Gupta et al. [12] present a technique for computing range queries using order-preserving hash functions. This system provides approximate answers to range queries, as opposed to the exact answers provided by P-Ring. The performance of the system proposed by Daskos et al. [9] depends on certain heuristics for insertion, and does not offer any performance guarantees. Sahin et al. [22] propose a caching scheme for queries, but no performance guarantees are provided for range queries which were not previously asked.

Skip Graphs [3] are a randomized structure based on skip lists. P-Tree [5] is a P2P index structure based on the B+ trees. Skip Graphs and P-Tree support routing of range queries, but, as opposed to P-Ring, they do not support multiple items per peer. Online Balancing [11] is a load balancing scheme for distributing items to peers with a provable bound of 4.24 for load imbalance with constant amortized insertion and deletion cost. The P-Ring Data Store achieves a better load balance with a factor of $2 + \epsilon$, while keeping the amortized insert/delete cost constant. Additionally, we also propose a new content router, the Hierarchical Ring. Mercury [4] is a randomized index structure determined by a sampling mechanism. P-Grid [2, 10] is a randomized trie-based index. Unlike P-Ring, Mercury and P-Grid provide only probabilistic guarantees even when the index is fully consistent. BATON [14] is a binary balanced tree with nodes distributed to peers in a P2P network. The P-Ring content router is more flexible, by allowing the application to choose higher values for d, the order of the HR, and thus to decrease the search cost, and the P-Ring Data Store provides provable guarantees on the load balance. BATON* [13] is extension of BATON, that provides search performance proportional to $log_d P$, but does not prove any guarantees on load balancing.

6. EXPERIMENTAL EVALUATION

We evaluate our system both using a simulation and a real implementation running on PlanetLab. We focus on two main aspects. First, we evaluate the performance of the P-Ring Data Store. As a baseline, we compare it with the hash-based Chord Data Store. In PlanetLab, we also compare it with Online Balancing [11]. Second, we evaluate the performance of the P-Ring Content Router, and compare it with Skip Graphs and Chord. We also consider the interaction between the two components in the presence of peer insertions and deletions (system "churn"). In all experiments, all the components of the index (Fault-tolerant Ring, Data Store, Replication, Content Router) are implemented and working, but we are only measuring the metrics of interest for the particular experiment.

6.1 Simulation Setup

We developed a simulator in C++ to evaluate the index structures. We implemented the P-Ring Data Store (Section 3.1), Hierarchical Ring (Section 4.1), Skip Graphs [3], and Chord [23]. Since Skip Graphs was originally designed for only a single item per peer, we extended it to use the P-Ring Data Store so that it could scale to multiple items per peer. For all the approaches, we implemented the same Fault Tolerant Ring [23] and Replication Manager [8].

We use three performance metrics: 1. *index message cost* - the average number of messages per minute (60 simulator time units) required for maintaining the index; 2. *index bandwidth cost* - the average number of bytes per minute required for maintaining the index; 3. *search cost* - the number of messages required to evaluate a range query, averaged over 100 random searches. Since the main variable component in the cost of range queries is finding the item with the smallest qualifying value (retrieving the other values has a fixed cost of traversing the relevant successor peers), we only report that cost. This also enables us to compare against Chord.

We varied the following parameters: *InsertionRate* (similarly, *DeletionRate*) is the rate of item insertions (deletions) into the system (default is 4 operations per second). *ItemInsertionPattern* (similarly, *ItemDeletionPattern*), specifies the skew in the values inserted (deleted) into the system. A value of *ip* for this parameter means that all insertions are localized within a fraction *ip* of the search key space (default is 1). *NumPeers* is the number of peers

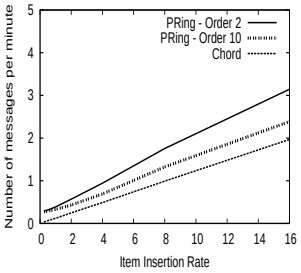

Figure 8. Item Insertion Rate

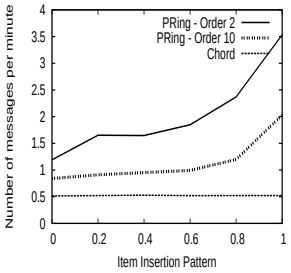

Figure 9. Insertion Pattern

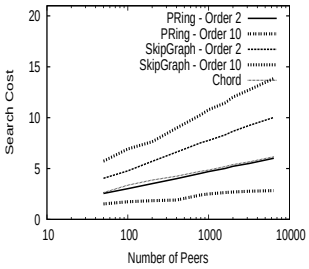

Figure 10. Number of Peers

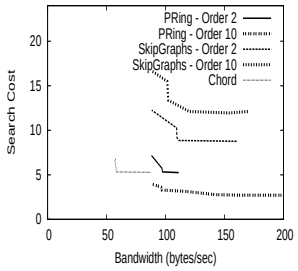

Figure 11. Performance

in the system (default is 2000). For each experiment we vary one parameter and use the default values for the rest. We first evaluate the Data Store and Content Router components in a stable system configuration (without peer joins/failures); we then investigate the effect of peer churn.

6.2 Experimental Results: Data Store

The performance of the Data Store partially depends on the performance of the Content Router (when inserting/deleting items or searching for helper peers). To isolate these effects, we fix the P-Ring Content Router to have orders 2 and 10 for this set of experiments and investigate different orders in next section. As a baseline for comparison, we use Chord, which is efficient due to hashing, but does not support range queries.

Varying Item Insertion Rate Figure 8 shows the index message cost as a result of varying *InsertionRate*. The message cost increases linearly with *InsertionRate* because each item insertion requires a search message to locate the peer that should store the item. The message cost increases faster for the P-Ring Data Store than for Chord because the P-Ring additionally needs to split and merge due to item skew, while Chord simply hashes the items. This difference quantifies the overhead of supporting range queries (using the P-Ring Data Store) as opposed to simple equality queries (using the Chord Data Store). Finally, the message cost for the P-Ring Data Store decreases as we use a Content Router of higher order because the search becomes more efficient with higher order Content Routers. The graph showing the index bandwidth cost is similar and is not shown. We also obtained similar results by varying *ItemDeletionPattern*.

Varying Item Insertion Pattern Figure 9 shows the index message cost as a result of varying *ItemInsertionPattern* from 0 - highly skewed distribution, to 1 - uniform distribution. For the Chord Data Store, as expected, we do not observe any significant variation in message cost. The message cost also remains relatively stable for P-Ring Data Store. This suggests that the P-Ring Data Store effectively manages item skew by splitting and merging as required. The surprising fact is that for P-Ring, the cost for uniform distribution is higher than for highly skewed distributions. The reason is that the cost of finding helper peers for split is included in the index cost. In skewed cases, most inserts happen close to 0, so most splits happen at peers close to 0. Since the helper peers are stored as items with search key value $\perp$ (see Section 3.2), they are also stored close to 0, so the search cost for finding a helper peer is very low, compared with the uniformly random case. The graph showing the index bandwidth cost is similar, and we also obtained similar results by varying *ItemDeletionPattern*.

6.3 Experimental Results: Content Router

We now investigate the performance of the P-Ring Content Router, and compare it with Skip Graphs and Chord.

Varying Number of Peers Figure 10 shows the search cost when varying the number of peers. As expected, the search cost increases logarithmically with the number of peers (note the logarithmic scale on the x-axis) for all the Content Routers. However, the search cost for the different Content Routers varies significantly. In particular, Skip Graphs has significantly worse search cost because the index structure of order d has search performance $O(d \times \log_d P)$ (where P is the number of peers in the system). In contrast, Chord has search cost $O(\log_2 P)$ and a P-Ring of order d has search cost $O(\log_d P)$. Due to the large base of the logarithm, the P-Ring of order 10 significantly outperforms the other index structures.

Varying Order Figure 12, Figure 13, and Figure 14 summarize the results of varying the order of the Hierarchical Ring. As expected, the search cost is $O(\log_d P)$. The index message cost decreases with order because there are fewer levels in the HR that need to be stabilized (recall that the number of levels in a HR of order d is $\log_d P$). However, the index bandwidth cost decreases slightly and then increases because, at higher orders, more information has to be transferred during index stabilization. Specifically, each stabilization message in a HR of order d has to transfer $O(d)$ information (the entries at one level). Hence, the total bandwidth requirement is $O(d \cdot \log_d P)$, which is consistent with the experimental results. This shows the tradeoff between maintenance cost and search cost - a higher value of d improves search but increases bandwidth requirements.

6.4 Experimental Results: System Churn

Figure 11 shows the effect of peer insertions and failures on index performance, for 4 insertions/failures per second (the results with other rates is similar), starting with a system of 2000 peers. The basic tradeoff is between search cost and index bandwidth cost. When the Content Router is stabilized at a high rate, bandwidth cost is high due to many stabilization messages, but the search cost is low since the Content Router is more consistent. On the other hand, when the Content Router is stabilized very slowly, the bandwidth cost decreases but the search cost increases. For P-Ring and Chord, the increase in search cost is small, even if the Content Router is temporarily inconsistent.

As shown in Figure 11, the P-Ring Content Router always dominates Skip Graphs due to its superior search performance. Chord outperforms P-Ring of order 2 because Chord does not have the overhead of dealing with splits and merges. However, P-Ring of order 10 offers a better search cost, albeit at a higher bandwidth cost, while also supporting range queries. We obtained similar results for search cost vs. index message cost.

6.5 Results from PlanetLab

We present preliminary results from our PlanetLab deployment. We implemented P-Ring, Online Balancing Fibbing algorithm [11], and Chord. The code base has more than 35,000 lines of C++ code and uses TCP/IP as communication protocol. We deployed our system on 50 random machines in PlanetLab [1], a network of computers distributed around the world.

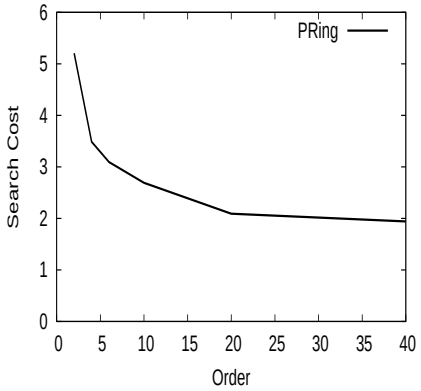

Figure 12. Search Cost vs. HR Order

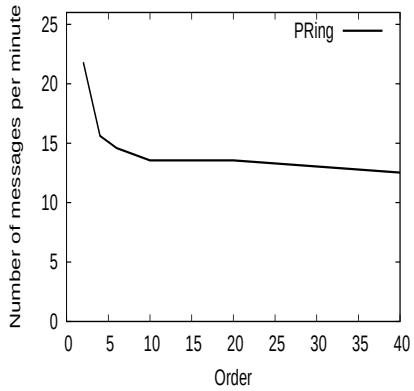

Figure 13. Message Cost

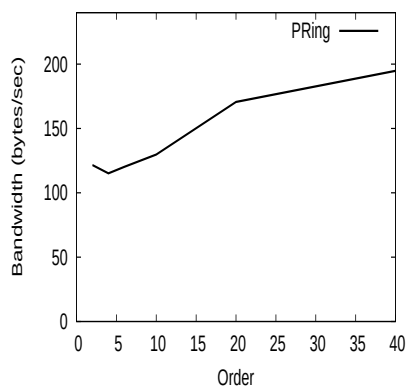

Figure 14. Bandwidth Cost

Item Churn In a first set of experiments, we study the performance of the system as items are inserted and deleted from the system (item churn). To see the effect of items insertions and deletions on the load balance, we start the system by inserting 50 peers and no data items. Then, we randomly insert/delete items in three phases: insert only, insert and delete, and delete only. In each phase we execute 2000 operations, at a rate of 1 operation/second. The items are inserted according to a Zipfian distribution with domain [1, 65536] and a skew parameter of 0.5. The items to be deleted are chosen uniformly at random from the existing items.

Figure 15 shows the load imbalance for P-Ring and Online Balancing during the course of the experiment. The load imbalance is measured each minute The load imbalance for P-Ring is almost always close to 2, while for Online Balancing, the load imbalance is above 2, but below 4.24 for the most part. The load imbalance is temporarily higher at the beginning of the insert phase, and at the end of the delete phase. There are two reasons for this. First, since we start and end with no data items, the average number of items is very low at the beginning and at the end. So inserting or deleting a few items can make a big difference in the load imbalance. Second, the ranges assigned to peers need to adapt to the change in data distribution. We see that the ranges adapt quickly, so after only a few minutes, the load imbalance is below the theoretical bound. This figure also shows that our method of estimating the storage factor for P-Ring gives correct results, as the load imbalance is indeed close to the theoretical one.

Figure 16 shows the average message cost for the maintenance of the Data Store component for Chord, P-Ring and Online Balancing. Similar trends were obtained for the bandwidth cost. We expected the maintenance cost for P-Ring and Online Balancing to be clearly higher than for Chord Data Store, due to the load-balancing operations. However, the differences in message costs are not big, especially during the insert/delete phase since there are very few re-balancing operations and the ranges have already adapted to the data distribution. Moreover, the item insert and delete message cost is similar for all structures (we used HR of order 2 for P-Ring and Online Balancing), and this cost is the major component of the maintenance cost. Note that for Online Balancing, the cost of maintaining an additional index on the load of the peers was *not* taken into consideration.

Figure 17 shows the evolution of load imbalance for P-Ring for different skew parameters. We see that regardless of how skewed the distribution is, the ranges adapt to the distribution using the re-balancing operations. A similar graph was obtained for Online Balancing.

Peer Churn In a second set of experiments, we study the effects of peer insertions and failures on load balancing. For these experiments we start the system by inserting 1 peer and 2000 data items with indexing attribute values following a Zipfian distribution with domain [1, 65536] and skew parameter 0.5. Then, peers randomly join/leave the system, in three phases: join only, join and leave, and leave only. In each phase we execute 50 operations, at the 0.02 operations/second rate.

Figure 18 shows the evolution of load imbalance for P-Ring and Online Balancing, as peers join and leave the system. Both algorithms adapt to the changes in the system, however the load imbalance is more variable than in the item churn case (see Figure 15). This is due to the fact that changes in the set of peers in the system, where each peer stores many data items, have a bigger impact on the number of items temporarily stored at each peer, and therefore on the load imbalance (when a peer fails/leaves the system, all the items previously stored by the failed peer will be recovered by its successor, since items are replicated, and that peer will temporarily be very overloaded; similarly, peer insertions could lead to underflow). As expected, the load imbalance is lower for P-Ring, than for Online Balancing.

Figure 19 shows the average message cost (results for bandwidth cost are similar) for maintaining the Data Store component for P-Ring, Online Balancing and Chord. The average number of messages is higher at the beginning as the 2000 items are inserted into the system, and there are only a few peers into the system. After the items are inserted, the average number of messages decreases. Figure 20 shows the details of the average message cost, with the highest values eliminated. Once all the items were inserted, the Data Store message cost for Chord is close to zero. This is because the Chord Data Store does not try to re-balance the ranges associated to the peers even during churn. The difference in cost between Chord and P-Ring and Online Balancing comes from the load balancing operations effectuated by P-Ring and Online Balancing, and represents the cost associated with providing extra functionality: explicit load balance, as opposed to the implicit load balance provided by hashing.

7. CONCLUSIONS

We have introduced P-Ring, a novel fault-tolerant P2P index structure that efficiently supports *both* equality and range queries in a dynamic P2P environment. P-Ring effectively balances items among peers even in the presence of skewed data insertions and deletions and provides provable guarantees on search performance. Our experimental evaluation shows that P-Ring outperforms existing index structures, sometimes even for equality queries, and that it maintains its excellent search performance with low maintenance cost in a dynamic P2P system.

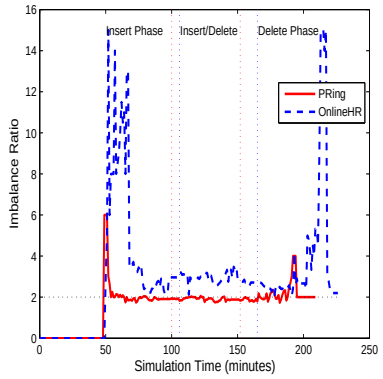

Figure 15. Load Imbalance

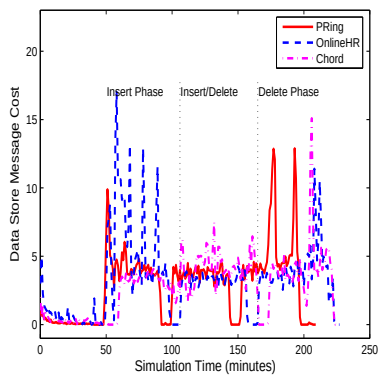

Figure 16. DS Cost

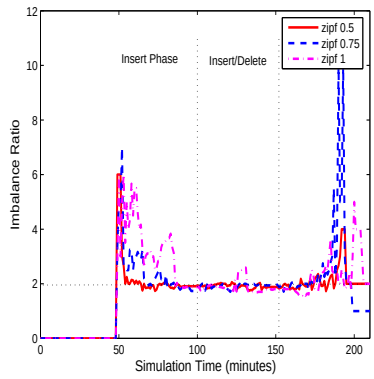

Figure 17. P-Ring Imbalance

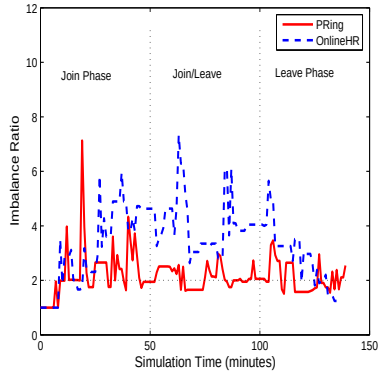

Figure 18. Load Imbalance - Churn

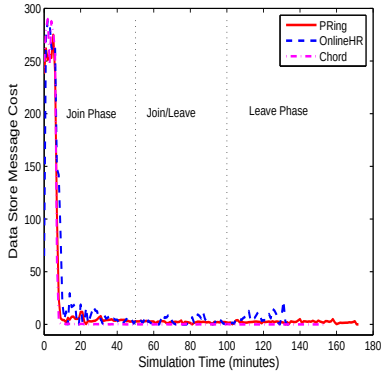

Figure 19. DS Cost - Churn

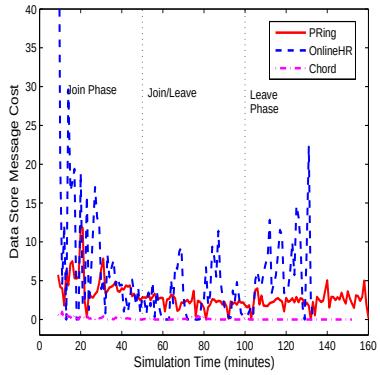

Figure 20. DS Cost Details - Churn

8. REFERENCES

[1] *PlanetLab website*, www.planet-lab.org.

[2] K. Aberer. P-grid: A self-organizing access structure for p2p information systems. In *CoopIS*, 2001.

[3] J. Aspnes and G. Shah. Skip graphs. In *SODA*, 2003.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4), 2004.

[5] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *WebDB*, 2004.

[6] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. An indexing framework for peer-to-peer systems. In *WWW (poster)*, 2004.

[7] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: An index structure for peer-to-peer systems. Technical report, Cornell University, http://www.cs.cornell.edu/database/Pepper/Pepper.htm, 2004.

[8] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[9] A. Daskos, S. Ghandeharizadeh, and X. An. Peper: A distributed range addressing space for p2p systems. In *DBISP2P*, 2003.

[10] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range-queries in trie-structured overlays. In *P2P Comp. 05*.

[11] P. Ganesan, M. Bawa, and H. Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[12] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *CIDR*, 2003.

[13] H. Jagadish, B. C. Ooi, K.-L. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *SIGMOD*, 2006.

[14] H. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *VLDB*, 2005.

[15] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasundaram. Guaranteeing correctness and availability in p2p range indices. In *SIGMOD*, 2005.

[16] W. Litwin, M.-A. Neimat, and D. A. Schneider. Lh* - linear hashing for distributed files. In *SIGMOD*, 1993.

[17] W. Litwin, M.-A. Neimat, and D. A. Schneider. Rp*: A family of order preserving scalable distributed data structures. In *VLDB*, 1994.

[18] D. B. Lomet. Replicated indexes for distributed data. In *PDIS*, 1996.

[19] S. Ratnasamy et al. A scalable content-addressable network. In *SIGCOMM*, 2001.

[20] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a dht. In *USENIX Annual Tech Conference*, 2004.

[21] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[22] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A p2p framework for caching range queries. In *ICDE*, 2004.

[23] I. Stoica et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[24] B. Y. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. In *Technical Report, U.C.Berkeley*, 2001.