Guaranteeing Correctness and Availability in P2P Range Indices

Prakash Linga, Adina Crainiceanu, Johannes Gehrke, Jayavel Shanmugasudaram
Cornell University
Ithaca, New York
{linga, adina, johannes, jai}@cs.cornell.edu

ABSTRACT

New and emerging P2P applications require sophisticated range query capability and also have strict requirements on query correctness, system availability and item availability. While there has been recent work on developing new P2P range indices, none of these indices guarantee correctness and availability. In this paper, we develop new techniques that can provably guarantee the correctness and availability of P2P range indices. We develop our techniques in the context of a general P2P indexing framework that can be instantiated with most P2P index structures from the literature. As a specific instantiation, we implement P-Ring, an existing P2P range index, and show how it can be extended to guarantee correctness and availability. We also quantitatively evaluate our techniques using a real distributed implementation.

1. INTRODUCTION

Peer-to-peer (P2P) systems have emerged as a promising paradigm for structuring large-scale distributed systems. The main advantages of P2P systems are scalability, fault-tolerance, and ability to reorganize in the face of dynamic changes to the system. A key component of a P2P system is a P2P index. A P2P index allows applications to store (value, item) pairs, and to search for relevant items by specifying a predicate on the value. Different applications have different requirements for a P2P index. We can characterize the index requirements of most P2P applications along the following three axes:

- **Expressiveness of predicates**: whether simple equality predicates suffice in a P2P index, or whether more complex predicates such as range predicates are required.
- **Query correctness**: whether it is crucial that the P2P index return all and only the data items that satisfy the predicate.
- **System and Item Availability**: whether it is crucial that the availability of the P2P index and the items stored in the index, are not reduced due to the reorganization of peers.

For example, simple file sharing applications only require support for equality predicates (to lookup a file by name), and do not have strict correctness and availability requirements (it is not catastrophic if a search occasionally misses a file, or if files are occasionally lost). Internet storage applications require only simple equality predicates, but have strict requirements on correctness and availability (so that data is not missed or lost). Digital library applications require complex search predicates such as range predicates (to search for articles within a date range), but do not have strict correctness and availability requirements. The most demanding applications are transaction processing and military applications, which require both complex range predicates (to search for objects within a region) and strong correctness/availability guarantees.

As an example, consider the Joint Battlespace Infosphere (JBI)[17], a military application that has high scalability and fault-tolerance requirements. One of the potential uses of the JBI is to track information objects, which could include objects in the field such as enemy vehicles. A natural way to achieve the desired scalability and fault-tolerance is to store such objects as (value,item) pairs in a P2P index, where the value could represent the geographic location of the object (in terms of its latitude and longitude), and the item could be a description of that object. Clearly, the JBI requires support for range queries in order to find objects in a certain region. The JBI also requires strong correctness guarantees (so that objects are not missed by a query) and availability guarantees (so that stored objects are not lost).

Current P2P indices, however, do not satisfy the above application needs: while there has been some work on devising P2P indices that can handle expressive range predicates [1, 2, 5, 6, 10, 12, 14, 15, 31], there has been little or no work on guaranteeing correctness and availability in such indices. Specifically, we are not aware of any P2P range index that *guarantees* that a query will not miss items relevant to a query. In fact, we shall later show scenarios whereby range indices [5, 6, 12] that are based on the Chord ring [32] (originally devised for equality queries) can miss query results for range queries, even when the index is operational. Similarly, we are not aware of any range index that can provide provable guarantees on system and item availability.

In this paper, we devise techniques that can provably guarantee query correctness, system availability and item availability in P2P range indices. At a high level, there are two approaches for guaranteeing correctness and availability. The first approach is to simply let the application handle the correctness and availability issues – this, for instance, is the approach taken by CFS [9] and PAST [30], which are applications built on top of the P2P equality indices Chord [32] and Pastry [29], respectively. However, this approach does not work in general for range indices because the application does not (and should not!) have control over various concurrent operations in a P2P range index, including index reorganization and peer failures. Moreover, this approach exposes low-level concurrency details to applications and is also very error-prone due

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA

to subtle concurrent interactions between system components.

We thus take the alternative approach of developing new correctness and availability primitives that can be directly implemented in a P2P index. Specifically, we build upon the P2P indexing framework proposed by Crainiceanu et al. [7], and embed novel techniques for ensuring correctness and availability directly into this framework. The benefits of this approach are that it abstracts away the dynamics of the underlying P2P system and provides applications with a consistent interface with provable correctness and availability guarantees. To the best of our knowledge, this is the first attempt to address these issues for both equality and range queries in a P2P index.

One of the benefits of implementing our primitives in the context of a P2P indexing framework is that our techniques are not just applicable to one specific P2P index, but are applicable to all P2P indices that can be instantiated in the framework, including [5, 6, 12]. As a specific instantiation, we implement P-Ring [6], a P2P index that supports both equality and range queries, and show how it can be extended to provide correctness and availability guarantees. We also quantitatively demonstrate the feasibility of our proposed techniques using a real distributed implementation of P-Ring.

The rest of the paper is organized as follows. In Section 2, we present some background material, and in Section 3, we outline our correctness and availability goals. In section 4 we present techniques for guaranteeing query correctness, and in Section 5, we outline techniques for guaranteeing system and item availability. In Section 6, we present our experimental results. In Section 7, we discuss related work, and we conclude in Section 8.

2. BACKGROUND

In this section, we first introduce our system model and the notion of a *history* of operations, which are used later in the paper. We then briefly review the indexing framework proposed by Crainiceanu et al.[7], and give an example instantiation of this framework for completeness. We use this instantiation in the rest of the paper to discuss problems with existing approaches and to illustrate our newly proposed techniques. We use the framework since it presents a clean way to abstract out different components of a P2P index, and it allows us to confine concurrency and consistency problems to individual components of the framework.

2.1 System Model

A *peer* is a processor with shared storage space and private storage space. The shared space is used to store the distributed data structure for speeding up the evaluation of user queries. We assume that each peer can be identified by a physical id (for example, its IP address). We also assume a fail-stop model for peer failures. A *P2P system* is a collection of peers. We assume there is some underlying network protocol that can be used to send messages reliably from one peer to another with known bounded delay. 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 each (data) item stored in a peer exposes a *search key value* from a totally ordered domain $\mathcal{K}$ that is indexed by the system. The search key value for an item i is denoted by $i.skv$. Without loss of generality, we assume that search key values are unique (duplicate values can be made unique by appending the physical id of the peer where the value originates and a version number; this transformation is transparent to users). Peers inserting items into the system can retain ownership of their items. In this case, the items are stored in the private storage partition of the peer, and only pointers to the items are inserted into the system.

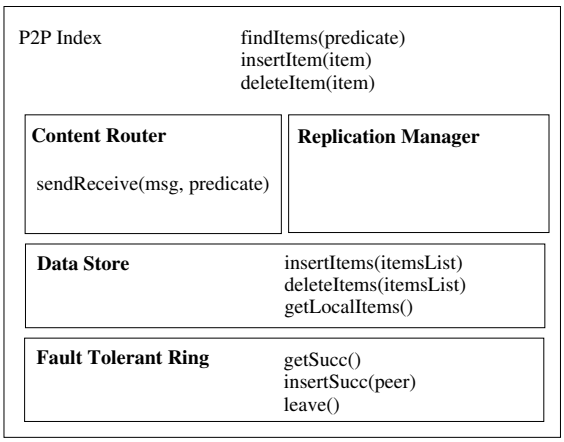

Figure 1. Indexing Framework

In the rest of the paper we make no distinction between items and pointers to items.

The queries we consider are range queries of the form $[lb, ub]$, $(lb, ub]$, $[lb, ub)$ or (lb, ub) where $lb, ub \in \mathcal{K}$. Queries can be issued at any peer in the system.

To specify and reason about the correctness and availability guarantees, we use the notion of a *history* of operations [4, 25].

Definition 1 (History $\mathcal{H}$): History $\mathcal{H}$ is a pair $(O, \leq)$ where O is a set of operations and $\leq$ is a partial order defined on these operations.

Conceptually, the partial order $\leq$ defines a *happened before* relationship among operations. If $op_1, op_2 \in O$ are two different operations in history $\mathcal{H}$, and $op_1 \leq op_2$, then intuitively, op_1 finished before op_2 started, i.e., op_1 happened before op_2. If op_1 and op_2 are not related by the partial order, then op_1 and op_2 could have been executed in parallel.

To present our results we also need the notion of a *truncated history* which is a history that only contains operations that happened before a certain operation.

Definition 2 (Truncated History $\mathcal{H}_o$): Given a history $\mathcal{H} = (O_\mathcal{H}, \leq_\mathcal{H})$ and an operation $o \in O_\mathcal{H}$, $\mathcal{H}_o = (O_{\mathcal{H}_o}, \leq_{\mathcal{H}_o})$ is a *truncated history* if $O_{\mathcal{H}_o} = \{o' \in O_\mathcal{H} | o' \leq_\mathcal{H} o\}$ and $\forall o_1, o_2 \in O_{\mathcal{H}_o}$ ($o_1 \leq_\mathcal{H} o_2 \Rightarrow o_1 \leq_{\mathcal{H}_o} o_2$).

2.2 The P2P Indexing Framework From [7]

A P2P index needs to reliably support the following operations: search, item insertion, item deletion, peers joining, and peers leaving the system. We now briefly survey the modularized indexing framework from [7], which is designed to capture most structured P2P indices. Figure 1 shows the components of the framework, and their APIs. The framework does not specify *implementations* for these components but only specifies *functional requirements*.

Fault Tolerant Torus. The Fault Tolerant Torus connects the peers in the system on a torus, and provides reliable connectivity among these peers even in the face of peer failures. For the purposes of this paper, we focus on a Fault Tolerant Ring (a one-dimensional torus). On a ring, for a peer p, we can define the *successor* $\texttt{succ}(p)$ (respectively, *predecessor* $\texttt{pred}(p)$) 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, then the ring will reorganize such that $\texttt{succ}(p_5) = p_2$, so the peers remain connected. In addition to maintaining successors, each peer p in the ring is associated with a value, $p.val$, from a totally ordered domain $\mathcal{PV}$. This value determines the position of a peer in the ring, and it increases clockwise around the ring (wrapping around

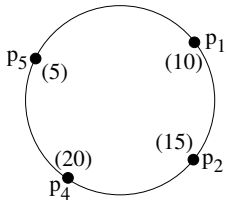

Figure 2. Ring

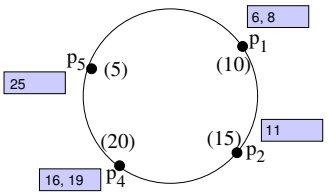

Figure 3. Data Store

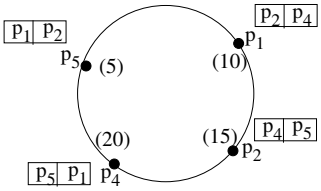

Figure 4. Chord Ring

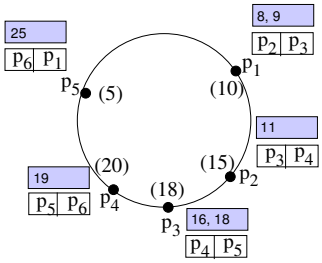

Figure 5. P-Ring Data Store

at the highest value). The values of the peers in Figure 2 are shown in paranthesis. The value of a peer is introduced only for ease of exposition and is not required in the formal definition of a ring.

Figure 1 shows the Fault Tolerant Ring API. When invoked on a peer p, p.`getSucc` returns the address of `succ`(p). p.`insertSucc`(p') makes p' the successor of p. p.`leave` allows p to gracefully leave the ring (of course, p can leave the ring without making this call due to a failure). The ring also exposes events that can be caught at higher layers, such as successor changes (not shown in the figure). An API method need not return right away because of locks and other concurrency issues. Each of the API methods is therefore associated with a start and an end operation. For example, $initLeave(p)$ is the operation associated with the invocation of the API method p.`leave()` and $leave(p)$ is the operation used to signal the end of this API method. All the operations associated with the initiation and completion of the API methods, as well as the operations associated with the events raised by the ring form a history called an *API Ring history*. The details can be found in [22].

Data Store. The Data Store is responsible for distributing and storing items at peers. The Data Store has a map $\mathcal{M}$ that maps the search key value $i.skv$ of each item i to a value in the domain $\mathcal{PV}$ (the domain of peer values). An item i is stored in a peer p such that $\mathcal{M}(i.skv) \in (\texttt{pred}(p).val, p.val]$. In other words, each peer p is responsible for storing data items mapped to a value between `pred`$(p).val$ and $p.val$. We refer to the range $(\texttt{pred}(p).val, p.val]$ as $p.range$. We denote the items stored at peer p as $p.items$.

Figure 3 shows an example Data Store that maps some search key values to peers on the ring. For example, peer p_4 is responsible for search key values 16 and 19. One of the main responsibilities of the Data Store is to ensure that the data distribution is uniform so that each peer stores about the same number of items. Different P2P indices have different implementations for the Data Store (e.g., based on hashing [32], splitting, merging and/or redistributing [6, 12]) for achieving this storage balance. As shown in Figure 1, the Data Store provides API methods to insert items into and delete items from the system. It also provides the API method p.`getLocalItems()` to get the items stored locally in peer p's Data Store.

As with the API Ring History, we can define the *API Data Store history* using the operations associated with the Data Store API methods. Given an API Data Store History $\mathcal{H}$ and a peer p, we use $range_{\mathcal{H}}(p)$ to denote $p.range$ in $\mathcal{H}$ and $items_{\mathcal{H}}(p)$ to denote $p.items$ in $\mathcal{H}$.

Replication Manager. The Replication Manager is responsible for reliably storing items in the system even in the presence of failures, until items are explicitly deleted. As an example, in Figure 5, peer p_1 stores items i_1 and i_2 such that $\mathcal{M}(i_1.skv) = 8$ and $\mathcal{M}(i_2.skv) = 9$. If p_1 fails, these items would be lost even though the ring would reconnect after the failure. The goal of the replication manager is to handle such failures for example by replicating items so that they can be "revived" even if peers fail.

Content Router. The Content Router is responsible for efficiently routing messages to relevant peers in the P2P system. As shown in the API (see Figure 1), the relevant peers are specified by a content-based predicate on search key values, and not by the physical peer ids. This abstracts away the details of storage and index reorganization from higher level applications.

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. As with the API Ring History and API Data Store History, we can define the *API Index History* using the operations associated with the Index API methods.

2.3 An Example Instantiation

We now discuss the instantiation of the above framework using P-Ring [6], an index structure designed for range queries in P2P systems. P-Ring uses the Fault Tolerant Ring of Chord and the Replication Manager of CFS, and only devises a new Data Store and a Content Router for handling data skew. While the full details of P-Ring are presented in [6], we concentrate only on features of P-Ring that are common to many P2P range query index structures from the literature [5, 6, 12]: splitting, merging, and redistributing in order to balance the number of items at each peer. We would like to emphasize that while we use P-Ring as a running example to illustrate query correctness, concurrency, and availability issues in subsequent sections, our discussion also applies to other P2P range indices proposed in the literature.

Fault Tolerant Ring. P-Ring uses the Chord Ring to maintain connectivity among peers [32]. The Chord Ring achieves fault-tolerance by storing a *list* of successors at each peer, instead of storing just a single successor. Thus, even if the successor of a peer p fails, p can use its successor list to identify other peers to reconnect the ring and to maintain connectivity. Figure 4 shows an example Chord Ring in which successor lists are of length 2 (i.e., each peer p stores `succ`(p) and `succ`$(\texttt{succ}(p))$ in its successor list). The successor lists are shown in the boxes next to the associated peers. Chord also provides a way to maintain these successor lists in the presence of failures by periodically *stabilizing* a peer p with its first live successor in the successor list. P-Ring also uses Chord to maintain connectivity.

Data Store. Ideally, we would like data items to be uniformly distributed among peers so that the storage load of each peer is about the same. Most existing P2P indices achieve this goal by *hashing* the search key value of an item, and assigning the item to a peer based on this hashed value. Such an assignment is, with high probability, very close to a uniform distribution of entries [28, 29, 32]. 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).

To solve this problem, range indices assign data items to peers

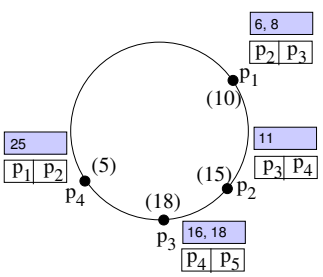

Figure 6. Data Store Merge

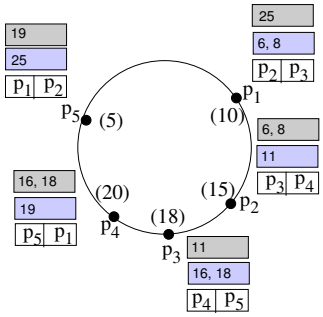

Figure 7. CFS Replication

directly based on their search key value (i.e., the map $\mathcal{M}$ is order-preserving, in the simplest case it is the identity function). In this case, the ordering of peer values is the same as the ordering of search key values, and range queries can be answered by scanning along the ring. The problem is that now, even in a stable P2P system with no peers joining or leaving, some peers might become overloaded or underloaded due to skewed item insertions and/or deletions. There is a need for a way to dynamically reassign and maintain the ranges associated to the peers. Range indices achieve this goal by `splitting`, `merging` and `redistributing` for handling item overflows and underflows in peers. Let us give an example in the context of P-Ring.

The P-Ring Data Store has two types of peers: *live* peers and *free* peers. Live peers can be part of the ring and store data items, while free peers are maintained separately in the system and do not store any data items.[1] The Data Store ensures that the number of items stored in each live peer is between `sf` and $2 \cdot$ `sf`, where `sf` is some *storage factor*, in order to balance storage between peers.

Whenever the number of items in a peer p's Data Store becomes larger than $2 \cdot$ `sf` (due to many insertions into $p.range$), it is said that an *overflow* occurred. In this case, p tries to *split* its assigned range (and implicitly its items) with a free peer, and to give a fraction of its items to the new peer. Whenever the number of entries in p's Data Store becomes smaller than `sf` (due to deletions from $p.range$), it is said that an *underflow* occurred. In this case, p tries to *merge* with its successor in the ring to obtain more entries. In this case, the successor either *redistributes* its items with p, or gives up its entire range to p and becomes a free peer.

As an illustration of a split, consider the Data Store shown in Figure 3. Assume that `sf` is 1, so each peer can have 1 or 2 entries. Now, when an item i such that $i.skv = 18$ is inserted into the system, it will be stored in p_4, leading to an overflow. Thus, $p_4.range$ will be split with a free peer, and p_4's items will be redistributed accordingly. Figure 5 shows the Data Store after the split, where p_4 split with the free peer p_3, and p_3 takes over part of the items p_4 was originally responsible for (the successor pointers in the Chord Ring are also shown in the figure for completeness). As an illustration of merge, consider again Figure 5 and assume that item i with $t.skv = 19$ is deleted from the system. In this case, there is an underflow at p_4, and p_4 merges with its successor, p_5 and takes over all of p_5's items; p_5 in turn becomes a free peer. Figure 6 shows the resulting system.

Replication Manager. P-Ring uses CFS Replication which works as follows. Consider an item i stored in the Data Store at peer p. The Replication Manager replicates i to k successors of p. In this way, even if p fails, i can be recovered from one of the successors of p. Larger values of k offer better fault-tolerance but have additional overhead. Figure 7 shows a system in which items are replicated with a value of $k = 1$ (the replicated values are shown in the top most box next to the peer).

Content Router. The P-Ring Content Router is based on idea of constructing a hierarchy of rings that can index skewed data distributions. The details of the content router are not relevant here.

[1] In the actual P-Ring Data Store, free peers also store data items temporarily for some live peers. The ratio of the number of items between any two peers can be bounded, but these details are not relevant in the current context.

3. GOALS

We now turn to the main focus of this paper: guaranteeing correctness and availability in P2P range indices. At a high level, our techniques enforce the following design goals.

- **Query Correctness:** A query issued to the index should return all and only those items in the index that satisfy the query predicate.
- **System Availability:** The availability of the index should not be reduced due to index maintenance operations (such as splits, merges, and redistributions).
- **Item Availability:** The availability of items in the index should not be reduced due to index maintenance operations (such as splits, merges, and redistributions).

While the above requirements are simple and natural, it is surprisingly hard to satisfy them in a P2P system. Thus, one approach is to simply leave these issues to higher level applications – this is the approach taken by CFS [9] and PAST [30], which are applications built on top of Chord [32] and Pastry [29], respectively, two index structures designed for equality queries. The downside of this approach is that it becomes quite complicated for application developers because they have to understand the details of how lower layers are implemented, such as how ring stabilization is done. Further, this approach is also error-prone because complex concurrent interactions between the different layers (which we illustrate in Section 4) make it difficult to devise a system that produces consistent query results. Finally, even if application developers are willing to take responsibility for the above properties, there are no known techniques for ensuring the above requirements for P2P range indices.

In contrast, the approach we take is to cleanly encapsulate the concurrency and consistency aspects in the different layers of the system. Specifically, we embed consistency primitives in the Fault Tolerant Ring and the Data Store, and provide handles to these primitives for the higher layers. With this encapsulation, higher layers and applications can simply use these APIs without having to explicitly deal with low-level concurrency issues or knowing how lower layers are implemented, while still being guaranteed query consistency and availability for range queries.

Our proposed techniques differ from distributed database techniques [20] in terms of scale (hundreds to thousands of peers, as opposed to a few distributed database sites), failures (peers can fail at any time, which implies that blocking concurrency protocols cannot be used), and perhaps most importantly, dynamics (due to unpredictable peer insertions and deletions, the location of the items is not known a priori and can change *during* query processing).

In the subsequent two sections, we describe our solutions to query correctness and system and item availability.

4. QUERY CORRECTNESS

We focus on query consistency for range queries (note that equality queries are a special case of range queries). We first formally define what we mean by query correctness in the context of the

indexing framework. We then illustrate scenarios where query correctness can be violated if we directly use existing techniques. Finally, we present our solutions to these problems. Detailed definitions and proofs for all theorems stated in this section can be found in [22].

4.1 Defining Correct Query Results

Intuitively, a system returns a correct result for a query Q if and only if the result contains all and only those items in the system that satisfy the query predicate. Translating this intuition into a formal statement in a P2P system requires us to define which items are "in the system"; this is more complex than in a centralized system because peers can fail, can join, and items can move between peers during the duration of a query. We start by defining an index P as a set of peers $P = \{p_1, \ldots, p_n\}$, where each peer is structured according to the framework described in Section 2.2. To capture what it means for an item to be in the system, we now introduce the notion of a *live item*.

Definition 3 (Live Item): An item i is *live* in API Data Store History $\mathcal{H}$, denoted by $live_{\mathcal{H}}(i)$, iff $\exists p \in P\ (i \in items_{\mathcal{H}}(p))$.

In other words, an item i is live in API Data Store History $\mathcal{H}$ iff the peer with the appropriate range contains i in its Data Store. Given the notion of a live item, we can define a correct query result as follows. We use $satisfies_Q(i)$ to denote whether item i satisfies query Q's query predicate.

Definition 4 (Correct Query Result): Given an API Data Store History $\mathcal{H} = (O_{\mathcal{H}}, \leq_{\mathcal{H}})$, a set R of items is a *correct query result* for a query Q initiated with operation o_s and successfully completed with operation o_e iff the following two conditions hold:

1. $\forall i \in R\ (\ satisfies_Q(i) \wedge \exists o \in O_{\mathcal{H}}\ (\ o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i)\ )\ )$
2. $\forall i\ (\ satisfies_Q(i) \wedge\ \forall o \in O_{\mathcal{H}}(o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i) \Rightarrow i \in R\ )\ )$.

The first condition states that only items that satisfy the query predicate and which were live at some time during the query evaluation should be in the query result. The second condition states that all items that satisfy the query predicate and which were live throughout the query execution must be in the query result.

4.2 Incorrect Query Results: Scenarios

Existing index structures for range queries evaluate a range query in two steps: (a) finding the peer responsible for left end of the query range, and (b) scanning along the ring to retrieve the items in the range. The first step is achieved using an appropriate Content Router, such as SkipGraphs [2] or the P-Ring [6] Content Router, and the related concurrency issues have been described and solved elsewhere in the literature [2, 6]. We thus focus on the second step (scanning along the ring) and show how existing techniques can produce incorrect results.

Scanning along the ring can produce incorrect query results due to two reasons. First, the ring itself can be temporarily inconsistent, thereby skipping over some live items. Second, even if the ring is consistent, concurrency issues in the Data Store can sometimes result in incorrect results. We now illustrate both of these cases using examples.

4.2.1 Inconsistent Ring

Consider the Ring and Data Store shown in Figure 5. Assume that item i with $\mathcal{M}(i.skv) = 6$ is inserted into the system. Since $p_1.range = (5, 10]$, i will be stored in p_1's Data Store. Now assume that p_1's Data Store overflows due to this insertion, and hence p_1 splits with a new peer p and transfers some of its items to p. The new state of the Ring and Data Store is shown in Figure 8. At this point, $p.range = (5, 6]$ and $p_1.range = (6, 10]$. Also, while p_5's successor list is updated to reflect the presence of p, the successor list of p_4 is not yet updated because the Chord ring stabilization proceeds in rounds, and p_4 will only find out about p when it next stabilizes with its successor (p_5) in the ring.

Now assume that p_5 fails. Due to the Replication Manager, p takes over the range $(20, 6]$ and adds the data item i such that $\mathcal{M}(i.skv) = 25$ into its Data Store. The state of the system at this time is now shown in Figure 9. Now assume that a search Q originates at p_4 for the range $(20, 9]$. Since $p_4.val$ is the lower bound of the query range, p_4 tries to forward the message to the first peer in its successor list (p_5), and on detecting that it has failed, forwards it to the next peer in its successor list (p_1). p_1 returns the items in the range $(6, 10]$, but the items in the range $(20, 6]$ are missed! (Even though all items in this range are live – they are in p's Data Store.) This problem arises because the successor pointers for p_4 are temporarily inconsistent during the insertion of p (they point to p_1 instead of p). Eventually, of course, the ring will stabilize and p_4 will point to p as its successor, but *before* this ring stabilization, query results can be missed.

At this point, the reader might be wondering whether a simple "fix" might address the above problem. Specifically, what if p_1 simply rejects the search request from p_4 (since p_4 is not p_1's predecessor) until the ring stabilizes? The problem with this approach is that p_1 does not know whether p has also failed, in which case p_4 is indeed p_1's predecessor, and it should accept the message. Again, the basic problem is that a peer does not have precise information about other peers in the system (due to the dynamics of the P2P system), and hence potential inconsistencies can occur. We note that the scenario outlined in Figure 9 is just one example of inconsistencies that can occur in the ring – rings with longer successor lists can have other, more subtle, inconsistencies (for instance, when p is not the direct predecessor of p_1).

4.2.2 Concurrency in the Data Store

We now show how concurrency issues in the Data Store can produce incorrect query results, *even if the ring is fully consistent*. We illustrate the problem in the context of a Data Store redistribute operation; similar problems arise for Data Store splits and merges.

Consider again the system in Figure 5 and assume that a query Q with query range $(10, 18]$ is issued at p_2. Since the lower bound of $p_2.range$ is the same as the lower bound of the query range, the sequential scan for the query range starts at p_2. The sequential scan operation first gets the data items in p_2's Data Store, and then gets the successor of p_2 in the ring, which is p_3. Now assume that the item i with $\mathcal{M}(i.skv) = 11$ is deleted from the index. This causes p_2 to become underfull (since it has no items left in its Data Store), and it hence redistributes with its successor p_3. After the redistribution, p_2 becomes responsible for the item i_1 with $\mathcal{M}(i_1.skv) = 16$, and p_3 is no longer responsible for this item. The current state of the index is shown in Figure 10.

Now assume that the sequential scan of the query resumes, and the scan operation propagates the scan to p_3 (the successor of p_2). However, the scan operation will miss item i_1 with $\mathcal{M}(i_1.skv) = 16$, even though i_1 satisfies the query range and was live throughout the execution of the query! This problem arises because of the concurrency issues in the Data Store – the range that p_2's Data Store was responsible for changed while p_2 was processing a query. Consequently, some query results were missed.

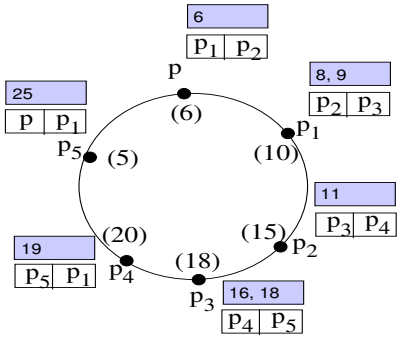

Figure 8. Peer p just inserted into the system

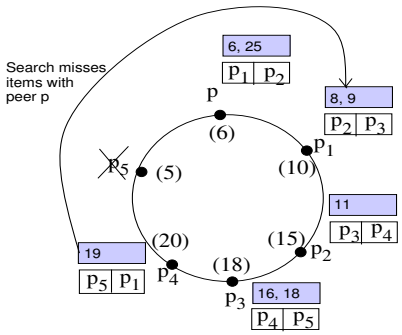

Figure 9. Incorrect query results: Search Q originating at peer p_4 misses items in p

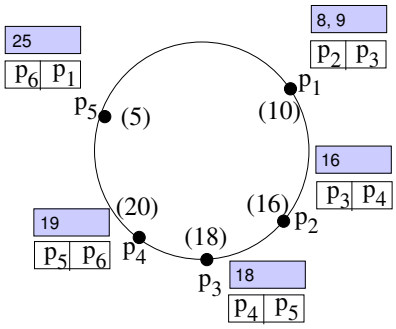

Figure 10. System after peer p_2 redistributes with peer p_3

4.3 Ensuring Correct Query Results

We now present solutions that avoid the above scenarios and provably guarantee that the sequential scan along the ring for range queries will produce correct query results. The attractive feature of our solution is that these enhancements are confined to the Ring and Data Store components of the architecture, and higher layers (both applications on top of the P2P system and other components of the P2P system itself) can be guaranteed correctness by accessing the components through the appropriate API. We first present a solution that addresses ring inconsistency, and then present a solution that addresses Data Store concurrency issues.

4.3.1 Handling Ring Inconsistency

As illustrated in Section 4.2.1, query results can be incorrect if a peer's successor list pointers are temporarily inconsistent (we shall formally define the notion of consistency soon). Perhaps the simplest way to solve this problem is to explicitly avoid this inconsistency by atomically updating the successor pointers of every relevant peer during each peer insertion. For instance, in the example in Section 4.2.1, we could have avoided the inconsistency if p_5's *and* p_4's successor pointers had been atomically updated during p's insertion. Unfortunately, this is not a viable solution in a P2P system because there is no easy way to determine the peers whose successor lists will be affected by an insertion since other peers can concurrently enter, leave or fail, and any cached information can become outdated.

To address this problem, we introduce a new method for implementing `insertSucc` (Figure 1) that ensures that successor pointers are always consistent even in the face of concurrent peer insertions and failures (peer deletions are considered in the next section). Our technique works asynchronously and does not require any up-to-date cached information or global co-ordination among peers. The main idea is as follows. Each peer in the ring can be in one of two states: `JOINING` or `JOINED`. When a peer is initially inserted into the system, it is in the `JOINING` state. Pointers to peers in the `JOINING` state need not be consistent. However, each `JOINING` peer transitions to the `JOINED` state in some bounded time. We ensure that the successor pointers to/from `JOINED` peers are always consistent. The intuition behind our solution is that a peer p remains in the `JOINING` state until all relevant peers know about p – it then transitions to the `JOINED` state. Higher layers, such as the Data Store, only store items in peers in the `JOINED` state, and hence avoid inconsistencies.

We now formally define the notion of consistent successor pointers. We then present our distributed, asynchronous algorithm for `insertSucc` that satisfies this property for `JOINED` peers.

4.3.1.1 Defining Consistent Successor Pointers

We first introduce some notation. Let $\mathcal{H}$ be a given API Ring History. This history induces a ring, denoted by $R_{\mathcal{H}}$. Let $P_{\mathcal{H}}$ be the set of live peers in `JOINED` state in the ring. $p.succList_{\mathcal{H}}$ is the successor list of peer p in $\mathcal{H}$. $p.succList_{\mathcal{H}}.length$ is the length (number of pointers) of $p.succList_{\mathcal{H}}$, and $p.succList_{\mathcal{H}}[i]$ ($0 \leq i < p.succList_{\mathcal{H}}.length$) refers to the i'th pointer in $succList$. We define $p.trimList_{\mathcal{H}}$ as the *trimmed* copy of $p.succList_{\mathcal{H}}$ with only pointers corresponding to live peers in `JOINED` state in $R_{\mathcal{H}}$.

Definition 5 (Consistent Successor Pointers): Given an API Ring History $\mathcal{H}$, the ring $R_{\mathcal{H}}$ induced by $\mathcal{H}$ has *consistent successor pointers* iff the following condition holds:

- $\forall p \in \mathcal{P}_{\mathcal{H}}$ ($\forall i$ ($0 \leq i < p.trimList_{\mathcal{H}}.length \Rightarrow succ_{\mathcal{H}}(p.trimList_{\mathcal{H}}[i]) = p.trimList_{\mathcal{H}}[i+1]$) $\wedge$ $succ_{\mathcal{H}}(p) = p.trimList_{\mathcal{H}}[0]$).

The above definition says that there are no peers in the ring between consecutive entries of $p.trimList$ i.e. p cannot have "missing" pointers to peers in the set $\mathcal{P}_{\mathcal{H}}$. In our example in Figure 8, the successor pointers are not consistent with respect to the set of all peers in the system because p_4 has a pointer to p_5 but not to p.

4.3.1.2 Proposed Algorithm

We first present the intuition behind our insert algorithm. Assume that a peer p' is to be inserted as the successor of a peer p. Initially, p' will be in the `JOINING` state. Eventually, we want p' to transition to the `JOINED` state, without violating the consistency of successor pointers. According to the definition of consistent successor pointers, the only way in which converting p' from the `JOINING` state to the `JOINED` state can violate consistency is if there exist `JOINED` peers p_x and p_y such that: $p_x.succList[i] = p$ and $p_x.succList[i+k] = p_y$ (for some $k > 1$) and for all $j, 0 < j < k$, $p_x.succList[i+j] \neq p'$. In other words, p_x has pointers to p and p_y but not to p' whose value occurs between $p.val$ and $p_y.val$.

Our algorithm avoids this case by ensuring that at the time p' changes from the `JOINING` state to the `JOINED` state, if p_x has pointers to p and p_y (where p_y's pointer occurs after p's pointer), then it also has a pointer to p'. It ensures this property by propagating the pointer to p' to all of p's predecessors until it reaches the predecessor whose last pointer in the successor list is p (which thus does not have a p_y that can violate the condition). At this point, it transitions p' from the `JOINING` to the `JOINED` state. Propagation of p' pointer is piggybacked on the Chord ring stabilization protocol, and hence does not introduce new messages.

Algorithm 1 : p_1.`insertSucc(`*`Peer`* p`)`

```
1: // Insert p into lists as a JOINING peer
2: writeLock succList, stateList
3: succList.push_front(p)
4: stateList.push_front(JOINING)
5: releaseLock stateList, succList
6: // Wait for successful insert ack
7: wait for JOIN ack; on ack do:
8: // Notify p of successful insertion and update lists
9: writeLock succList, stateList
10: Send a message to p indicating it is now JOINED
11: stateList.update_front(JOINED)
12: succList.pop_back(), stateList.pop_back()
13: releaseLock stateList, succList
```

Algorithm 2 : `Ring Stabilization`

```
1: // Update lists based on successor's lists
2: readLock succList, stateList
3: get succList/stateList from first non-failed p_s in succList
4: upgradeWriteLock succList, stateList
5: succList = p_s.succList; stateList = p_s.stateList
6: succList.push_front(p_s)
7: stateList.push_front(JOINED)
8: succList.pop_back(), stateList.pop_back()
9: // Handle JOINING peers
10: listLen = succList.length
11: if stateList[listLen − 1] == JOINING then
12:    succList.pop_back(); stateList.pop_back()
13: else if stateList[listLen − 2] == JOINING then
14:    Send an ack to succList[listLen − 3]
15: end if
16: releaseLock stateList, succList
```

Algorithms 1 and 2 show the pseudocode for the `insertSucc` method and the modified ring stabilization protocol, respectively. In the algorithms, we assume that in addition to $succList$, each peer has a list called $stateList$ which stores the state (`JOINING` or `JOINED`) of the corresponding peer in $succList$. We walk through the algorithms using an example.

Consider again the example in Figure 5, where p is to be added as a successor of p_5. The `insertSucc` method is invoked on p_5 with a pointer to p as the parameter. The method first acquires a write lock on $succList$ and $stateList$, inserts p as the first pointer in $p_5.succList$ (thereby increasing its length by one), and inserts a corresponding new entry into $p_5.stateList$ with value `JOINING` (lines $2-4$ in Algorithm 1). The method then releases the locks on $succList$ and $stateList$ (line 5) and blocks waiting for an acknowledgment from some predecessor peer indicating that it is safe to transition p from the `JOINING` state to the `JOINED` state (line 7). The current state of the system is shown in Figure 11 (`JOINING` list entries are marked with a "*").

Now assume that a ring stabilization occurs at p_4. p_4 will first acquire a read lock on its $succList$ and $stateList$, contact the first non-failed entry in its successor list, p_5, to get p_5's $succList$ and $stateList$ (lines $2-3$ in Algorithm 2). p_4 then acquires a write lock on its $succList$ and $stateList$, and copies over the $succList$ and $stateList$ it obtained from p_5 (lines $4-5$). p_4 then inserts p_5 as the first entry in $succList$ (increasing its length by 1) and also inserts the corresponding state in $stateList$ (the state will always be `JOINED` because `JOINING` nodes do not respond to ring stabilization requests). p_4 then removes the last entries in $succList$ and $stateList$ (lines $6-8$) to ensure that its lists are of the same length as p_5's lists. The current state of the system is shown in Figure 12.

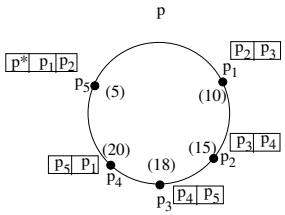

Figure 11. After p_5.insertSucc call

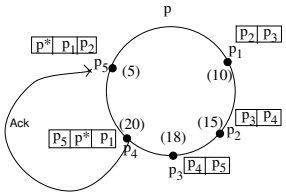

Figure 12. Propagation and final ack

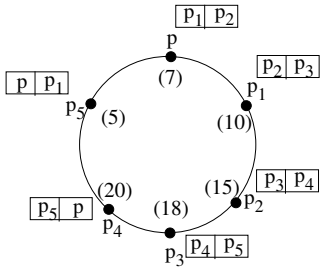

Figure 13. Completed `insertSucc`

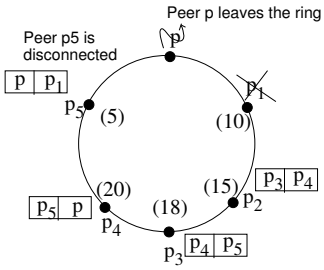

Figure 14. Naive merge leads to reduced reliability

p_4 then checks whether the state of the last entry is `JOINING`; in this case it simply deletes the entry (lines $11-12$) because it is far enough from the `JOINING` node that it does not need to know about it (although this case does not arise in our current scenario for p_4). p_4 then checks if the state of the penultimate peer (p) is `JOINING` – since this is the case in our scenario, p_4 sends a acknowledgment to the peer preceding the penultimate peer in the successor list (p_5) indicating that p can be transitioned from `JOINING` to `JOINED` since all relevant predecessors know about p (lines $13-14$). p_4 then releases the locks on its lists (line 16).

The `insertSucc` method of p_5, on receiving a message from p_4, first send a message to p indicating that it is now in the `JOINED` state (line 10). p_5 then changes the state of its first list entry (p) to `JOINED` and removes the last entries from its lists in order to shorten them to the regular length (lines $11-12$). The final state after p is inserted into the ring and multiple ring stabilizations have occurred is shown in Figure 13.

One optimization we implement for the above method is to *proactively* contact the predecessor in the ring whenever `insertSucc` is in progress, to trigger ring stabilization. This expedites the operation since it is no longer limited by the frequency of the ring stabilization process.

We can define a *PEPPER Ring History* to capture our implementation of the ring API, including the operations in Algorithms 1 and 2. We can prove the following theorem.

Theorem 1 (Consistent Successor Pointers): *Given a PEPPER Ring History $\mathcal{PH}$, the ring $R_{\mathcal{PH}}$ induced by $\mathcal{PH}$ has consistent successor pointers.*

4.3.2 Handling Data Store Concurrency

Recall from the discussion in Section 4.2.2 that even if the ring is fully consistent, query results can be missed due to concurrency issues at the Data Store. Essentially, the problem is that the range of a peer can change while a query is in progress, causing the query to miss some results. How do we shield the higher layers from the concurrency details of the Data Store while still ensuring correct query results?

Our solution to this problem is as follows. We introduce a new API method for the Data Store called `scanRange`. This method

has the following signature: `scanRange(lb, ub, handlerId, param)`, where (1) *lb* is the lower bound of the range to be scanned, (2) *ub* is the upper bound of the range to be scanned, (3) *handlerId* is the id of the handler to be invoked on every peer p such that $p.range$ intersects $[lb, ub]$ (i.e., p's range intersects the scan range), and (4) *param* is the parameter to be passed to the handlers. The `scanRange` method should be invoked on the Data Store of the peer p_1 such that $lb \in p_1.range$ (i.e., the first peer whose range intersects the scan range). The start and end operations associated with `scanRange` are $initScanRange_i(p_1, lb, ub)$ and $doneScanRange_i(p_n, lb, ub)$ for some $i \in \mathcal{N}$. The index i is used to distinguish multiple invocations of the API method with the same signature. The `scanRange` method causes the appropriate handler to be invoked on every peer p such that $p.range$ intersects $[lb, ub]$. $scanRange_i(p, p_1, r)$ is the operation in the API Data Store History that is associated with the invocation of the appropriate handler at peer p. Here, r is the subset of $p.range$ that intersects with $[lb, ub]$.

`scanRange` handles all the concurrency issues associated with the Data Store. Consequently, higher layers do not have to worry about changes to the Data Store while a scan is in progress. Further, since `scanRange` allows applications to register their own handlers, higher layers can customize the scan to their needs (we shall soon show how we can collect range query results by registering appropriate handlers).

We now introduce some notation before we define the notion of *scanRange correctness*. We use $scanOps(i)$ to denote the set of $scanRange_i(p, p_1, r)$ operations associated with the i^{th} invocation of `scanRange`. We use $rangeSet(i) = \{r | \exists p_1, p_2$ $scanRange_i(p_1, p_2, r) \in scanOps(i)\}$ to denote the set of ranges reached by `scanRange`. We use $r_1 \bowtie r_2$ to denote that range r_1 overlaps with range r_2 and we use $r_1 \cup r_2$ to denote the union of range r_1 with range r_2.

We can define *scanRange correctness* as follows:

Definition 6 (scanRange Correctness): An API Data Store History $\mathcal{H} = (O_\mathcal{H}, \leq_\mathcal{H})$ is said to satisfy *scanRange correctness* iff $\forall i \in \mathcal{N}\ \forall lb, ub\ \forall p_1 \in \mathcal{P}\ o_e = doneScanRange_i(p_1, lb, ub) \in O_\mathcal{H} \Rightarrow$

1. $o_s = initScanRange_i(p_1, lb, ub) \leq_\mathcal{H} o_e$
2. $\forall o \in scanOps(i)\ \forall p\ \forall r\ o = scanRange_i(p, p_1, r)$ $\Rightarrow o_s \leq_\mathcal{H} o \leq_\mathcal{H} o_e \wedge r \subseteq range_{\mathcal{H}_o}(p)$
3. $\forall o_l, o_m \in scanOps(i)\ o_l \neq o_m \wedge \forall p_l, p_m\ \forall r_l, r_m\ o_l =$ $scanRange_i(p_l, p_1, r_l) \wedge o_m = scanRange_i(p_m, p_1, r_m)$ $\Rightarrow \neg(o_l \bowtie o_m)$
4. $[lb, ub] = \cup_{r \in rangeSet(i)}(r)$

Condition 1 states that the initiate operation for `scanRange` should occur before the completion operation. Condition 2 states that range r used to invoke the handler at peer p is a subset of p's range. Condition 3 states that ranges r_l and r_m used to invoke the handlers at distinct peers p_l and p_m, respectively, are non-overlapping. Finally, condition 4 states that the union of all ranges used to invoke the handlers is $[lb, ub]$.

4.3.2.1 Implementing scanRange

We present now our implementation for the `scanRange` API method. Algorithm 3 shows the pseudocode for the `scanRange` method executed at a peer p. The method first acquires a read lock on the Data Store $range$ (to prevent it from changing) and then checks to make sure that $lb \in p.range$, i.e., p is the first peer in the range to be scanned (lines 1-2). If the check fails, `scanRange` is aborted (lines 3-4). If the check succeeds, then the helper method `processHandler` is invoked.

Algorithm 3 : $p.scanRange(lb, ub, handlerId, param)$

```
1: readLock range
2: if lb ∉ p.range then
3:    // Abort scanRange
4:    releaseLock range
5: else
6:    // p is the first peer in scan range
7:    p.processHandler(r, handlerId, param)
8: end if
```

Algorithm 4 : $p.processHandler(lb, ub, handlerId, param)$

```
1: // Invoke appropriate handler with relevant range r
2: Get handler with id handlerId
3: r = [lb, ub] ∩ p.range
4: newParam = handler.handle(r, param)
5: // Forward to successor if required
6: if ub ∉ p.range then
7:    p_succ = p.ring.getSucc()
8:    p_succ.processScan(lb, ub, handlerId, newParam)
9: end if
10: releaseLock range
```

`processHandler` (Algorithm 4) first invokes the appropriate handler for the scan (lines 1-3), and then checks to see whether the scan has to be propagated to p's successor (line 4). If so, it invokes the `processScan` method on p's successor.

Algorithm 5 shows the code that executes when p_{succ}.`processScan` is invoked by p.`processHandler`. `processScan` *asynchronously* invokes the `processHandler` method on p_{succ}, and returns. Consequently, p holds on to a lock on its range only until p_{succ} locks its range; once p_{succ} locks its range, p can release its lock, thereby allowing for more concurrency. Note that p can later split, merge, or redistribute, but this will not produce incorrect query results since the scan has already finished scanning the items in p.

We now illustrate the working of these algorithms using an example. Assume that `scanRange`$(10, 18, h_1, param_1)$ is invoked in p_2 in Figure 5. p_2 locks its range in `scanRange` (to prevent p_2's range from changing), invokes the handler corresponding to h_1 in `processHandler`, and then invokes `processScan` on p_3. p_3 locks its range in `processScan`, asynchronously invokes `processHandler` and returns. Since p_3.`processScan` returns, p_2 can now release its lock and participate in splits, merges, or redistributions. However, p_3 holds onto a lock on its range until p_3 handler is finished executing. Thus, the algorithms ensure that a peer's range does not change during a scan, but releases locks as soon as the scan is propagated to the peer's successor, for maximum concurrency.

We can define a *PEPPER Data Store History* to capture our implementation of the Data Store API augmented with the new operation `scanRange` . We can prove the following correctness theorem.

Theorem 2 (scanRange Correctness): *Any PEPPER Data Store History satisfies the scanRange correctness property*.

Using the `scanRange` method, we can easily ensure correct results for range queries by registering the appropriate handler. Algorithm 6 shows the algorithm for evaluating range queries. lb and ub represent the lower and upper bounds of the range to be scanned, and pid represents the id of the peer to which the final result is to be sent. As shown, the algorithm simply invokes the `scanRange` method with parameters lb, ub, the id of the range query handler, and a parameter for that handler. The id of the peer pid that the result should be sent to is passed as a parameter to the range query

Algorithm 5 : $p.processScan(lb, ub, handlerId, param)$

1: readLock $range$
2: Invoke $p.processHandler(lb, ub, handlerId, param)$ asynchronously
3: return

Algorithm 6 : $p.rangeQuery(lb, ub, pid)$

1: // Initiate a scanRange
2: p.scanRange($lb, ub, rangeQueryHandlerId, pid$)

handler. The range query handler (Algorithm 7) invoked with range r at a peer p works as follows. It first gets the items in p's Data Store that are in range r and hence satisfy the query result (lines 1-2). Then, it sends the items and the range r to the peer pid (line 3).

Using the above implementation of a range query, the inconsistency described in Section 4.2.2 cannot occur because p_2's range cannot change (and hence redistribution cannot happen) when the search is still active in p_2. We can prove the following correctness theorem:

Theorem 3 (Search Correctness): *Given a PEPPER Data Store History $\mathcal{PH}$, all query results produced in $\mathcal{PH}$ are* correct *(as per the definition of correct query results in Section 4.1).*

5. SYSTEM AND ITEM AVAILABILITY

We now address system availability and item availability issues. Intuitively, ensuring system availability means that the availability of the index should not be reduced due to routine index maintenance operations, such as splits, merges, and redistributions. Similarly, ensuring item availability means that the availability of items should not be reduced due to maintenance operations. Our discussion of these two issues is necessarily brief due to space constraints, and we only illustrate the main aspects and sketch our solutions.

5.1 System Availability

An index is said to be *available* if its Fault Tolerant Ring is connected. The rationale for this definition is that an index can be operational (by scanning along the ring) so long as its peers are connected. The Chord Fault Tolerant Ring provides strong availability guarantees when the only operations on the ring are peer insertions (splits) and failures [32]. These availability guarantees also carry over to our variant of the Fault Tolerant Ring with the new implementation of `insertSucc` described earlier because it is a stronger version of the Chord's corresponding primitive (it satisfies all the properties required for the Chord proofs). Thus, the only index maintenance operation that can reduce the availability of the system is the merge operation in the Data Store, which translates to the `leave` operation in the Fault Tolerant Ring. Note that the redistribute operation in the Data Store does not affect the ring connectivity.

We show that a naive implementation of `leave`, which is simply removing the merged peer from the ring, reduces system availability. We then sketch an alternative implementation for the `leave` that provably does not reduce system reliability. Using this new implementation, the Data Store can perform a merge operation without knowing the details of the ring stabilization, while being guaranteed that system availability is not compromised.

Naive `leave` Reduces System Availability: Consider the system in Figure 13 in which the length of the successor list of each peer is 2. Without a `leave` primitive, this system can tolerate one failure per peer stabilization round without disconnecting the ring (since at most one of a peer's two successor pointers can become invalid before the stabilization round). We now show that in the presence

Algorithm 7 : $p.rangeQueryHandler(r, pid)$

1: // Get results from p's Data Store
2: Find $items$ in p's Data Store in range r
3: Send $< items, r >$ to peer pid

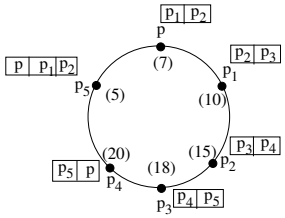

Figure 15. Controlled leave of peer p

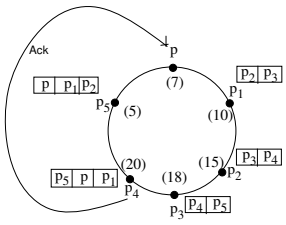

Figure 16. Final ack received at peer p. Peer p is good to go.

of the naive `leave`, a single failure can disconnect the ring. Thus, `leave` reduces the availability of the system. Assume that `leave` is invoked on p, and p immediately leaves the ring. Now assume that p_1 fails (this is the single failure). The current state of the system is shown in Figure 14, and as we can see, the ring is disconnected since none of p_5's successor pointers point to peers in the ring.

Solution Sketch: The reason the naive implementation of `leave` reduced availability is that pointers to the peer p leaving the ring become invalid. Hence, the successor lists of the peers pointing to p effectively decreases by one, thereby reducing availability. To avoid this problem, our solution is to increase the successor list lengths of all peers pointing to p by one. In this way, when p leaves, the availability of the system is not compromised. As in the `insertSucc` case, we piggyback the lengthening of the successor lists on the ring stabilization protocol. This is illustrated in the following example.

Consider Figure 13 in which `leave` is invoked on p. During the next ring stabilization, the predecessor of p, which is p_5, increases its successor list length by 1. The state of the system is shown in Figure 15. During the next ring stabilization, the predecessor of p_5, which is p_4, increases its successor list length by 1. Since p_4 is the last predecessor that knows about p, p_4 sends a message to p indicating that it is safe to leave the ring. The state of the system at this point is shown in Figure 16. It is easy to see that if p leaves the ring at this point, a single failure cannot disconnect the ring, as in was the case in the previous example. We can formally prove that the new algorithm for `leave` does not reduce the availability of the system.

5.2 Item Availability

We first formalize the notion of item availability in a P2P index.

We represent the successful insertion of an item i at peer p with operation $insertItem(i, p)$ and deletion of an item i' at peer p' with operation $deleteItem(i', p')$.

Definition 7 (Item Availability): Given an API Index History $\mathcal{H}$, an index P is said to preserve *item availability* iff
$\forall i \; (\; \exists p \in P \; (\; insertItem(i, p) \in O_{\mathcal{H}} \;) \; \wedge$
$\nexists p' \in P \; (\; deleteItem(i, p') \in O_{\mathcal{H}} \;) \Rightarrow live_{\mathcal{H}}(i) \;).$

In other words, if item i has been inserted but not deleted wrt to *API Index history* $\mathcal{H}$ then i is a live item.

The CFS Replication Manager, implemented on top of the Chord Ring provides strong guarantees [9] on item availability when the only operations on the ring are peer insertions and failures, and these carry over to our system too. Thus, the only operation that

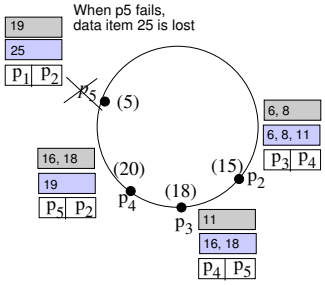

Figure 17. Peer p_5 fails causing loss of item 25

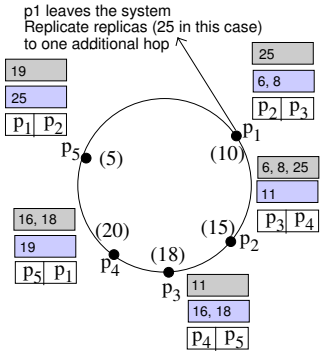

Figure 18. Replicate item 25 one additional hop.

could compromise item availability is the `leave` operation invoked on a merge. We now show that using the original CFS Replication Manager in the presence of merges does compromise item availability. We then describe a modification to the CFS Replication Manager and its interaction with the Data Store that ensures the original guarantees on item availability.

Scenario that Reduces Item Availability: Consider the system in Figure 7. The top box associated with each peer represents the items replicated at that peer (CFS replicates items along the ring). In this example, each item is replicated to one successor along the ring; hence, the system can tolerate one failure between replica refreshes. We now show how, in the presence of Data Store merges, a single failure can compromise item availability. Assume that peer p_1 wishes to merge with p_2 in Figure 7. p_1 thus performs an `leave` operation, and once it is successful, it transfers its Data Store items to p_2 and leaves the system. The state of the system at this time is shown in Figure 17. If p_5 fails at this time (this is the single failure), the item i such that $\mathcal{M}(i.skv) = 25$ is lost.

Solution Sketch: The reason item availability was compromised in the above example is because when p_1 left the system, the replicas it stored were lost, thereby reducing the number of replicas for certain items in the system. Our solution is to replicate the items stored in the merging peer p's Replication Manager for one additional hop before p leaves the system. This is illustrated in Figure 18, where before p_1 merges with p_2, it creates one more replica for items in its Data Store and Replication Manager, at one additional peer. When p_1 finally merges with p_2 and leaves the system, the number of replicas is not reduced, thereby preserving item availability. We can prove that the above scheme preserves item availability even in the presence of concurrent splits, merges, and redistributions.

6. EXPERIMENTAL EVALUATION

We had two main goals in our experimental evaluation: (1) to demonstrate the feasibility of our proposed query correctness and availability algorithms in a dynamic P2P system, and (2) to measure the overhead of our proposed techniques. Towards this goal, we implemented the P-Ring index, along with our proposed correctness and availability algorithms, in a real distributed environment with concurrently running peers. We used this implementation to measure the overhead of each of our proposed techniques as compared to the naive approach, which does not guarantee correctness or availability.

6.1 Experimental Setup

We implemented the P-Ring index as an instantiation of the indexing framework (Section 2.3). The code was written in C++ and all experiments were run on a cluster of workstations, each of which had 1GHz processor, 1GB of main memory and at least 15GB of disk space. All experiments were performed with 30 peers running concurrently on 10 machines (with 3 peers per machine). The machines were connected by a local area network.

We used the following default parameter values for our experiments. The length of the Chord Fault-Tolerant Ring successor list was 4 (which means that the ring can tolerate up to 3 failures without being disconnected if the ring is fully consistent). The ring stabilization period was 4 seconds. We set the storage factor of the P-Ring Data Store to be 5, which means that it can hold between 5 and 10 data items. The replication factor in the Replication Manager is 6, which means that each item is replicated 6 times. We vary these parameters too in some of the experiments.

We ran experiments in two modes of the system. The first mode was the *fail-free* mode, where there were no peers failures (although peers are still dynamically added and splits, merges, and redistributes occur in this state). The second was the *failure* mode, where we introduced peer failures by killing peers. For both modes, we added peers at a rate of one peer every 3 seconds, and data items were added at the rate of 2 items per second. We also vary the rate of peer failures in the failure mode.

6.2 Implemented Approaches

We implemented and evaluated all four of the techniques proposed in this paper. Specifically, we evaluate (1) the *insertSucc* operation that guarantees ring consistency, (2) the *scanRange* operation that guarantees correct query results, (3) the *leave* operation that guarantees system availability, and (4) the *replication to additional hop* operation that guarantees item availability. For *scanRange*, we implemented a synchronous version where the *processHandler* is invoked synchronously at each peer (see Algorithm 5).

One of our goals was to show that the proposed techniques actually work in a real distributed dynamic P2P system. The other goal was to compare each solution with a naive approach (that does not provide correctness or availability guarantees). Specifically, for the *insertSucc* operation, we compare it with the naive *insertSucc*, where the joining peer simply contacts its successor and becomes part of the ring. For the *scanRange* operation, we compare it with the naive range query method whereby the application explicitly scans the ring without using the *scanRange* primitive. For the *leave* operation, we compare with the naive approach where the peer simply leaves the system without notifying other peers. Finally, for the *replication to additional hop* operation, we compare with the naive approach without additional replication.

6.3 Experimental Results

We now present our experimental results. We first present results in the fail-free mode, and then present results in the failure mode.

6.3.1 Evaluating insertSucc

In this section we quantify the overhead of our *insertSucc* when compared to the naive *insertSucc*. The performance metric used is the time to complete the operation; this time is averaged over all such operations in the system during the run of the experiment.

We vary two parameters that affect the performance of the operations. The first parameter is the length of the ring successor list. The longer the list, the farther *insertSucc* has to propagate information before it can complete. The second is the ring stabilization period. The longer the stabilization period, the slower information about joining peers propagates due to stabilization.

Figure 19 shows the effect of varying the ring successor list

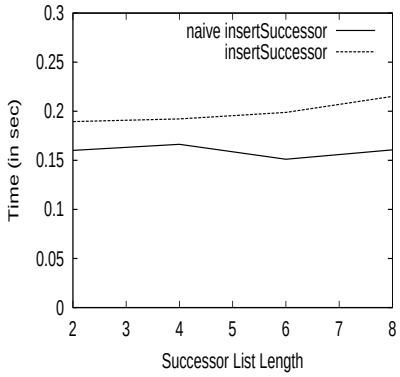

Figure 19. Overhead of insertSucc

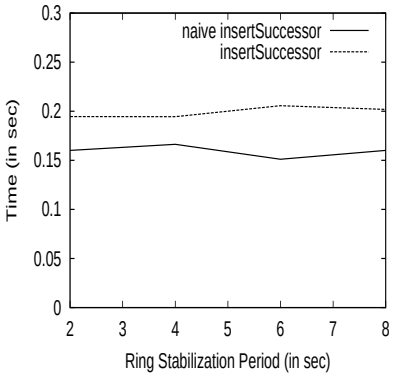

Figure 20. Overhead of insertSucc

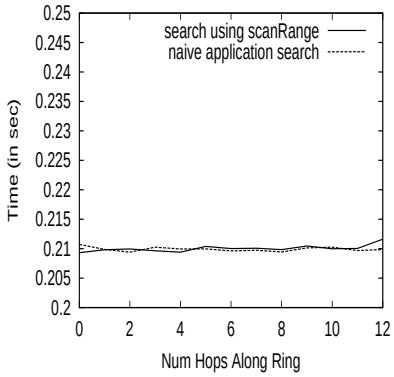

Figure 21. Overhead of scanRange

length. There are several aspects to note about this figure. First, the time for our *insertSucc* increases linearly with the successor list length, while the time for the naive *insertSucc* remains constant. This is to be expected because the naive *insertSucc* only contacts the successor, while our *insertSucc* propagates information to as many predecessors as the length of the successor list. Second, perhaps surprisingly, the rate of increase of the time for our *insertSucc* operation is very small; this can be attributed to the optimization discussed in Section 4.3.1, where we proactively contact predecessors instead of only relying on the stabilization. Finally, an encouraging result is that the cost of our *insertSucc* is of the same ball park as that of the naive *insertSucc*; this means that users do not pay too high a price for consistency.

Figure 20 shows the result of varying the ring stabilization frequency. The results are similar to varying the successor list length. Varying the ring stabilization period also has less of an effect on our *insertSucc* because of our optimization of proactively contacting predecessors.

6.3.2 *Evaluating* scanRange

In this section, we investigate the overhead of using *scanRange* when compared to the naive approach of the application scanning the range by itself. Since the number of messages needed to complete the operation is the same for both approaches, we used the elapsed time to complete the range search as the relevant performance metric. We varied the size of the range to investigate its effect on performance, and averaged the elapsed time over all the searches requiring the same number of hops along the ring. Each peer generates searches for ranges of different sizes, and we measured the time needed to process the range search, once the first peer with items in the search range was found. This allows us to isolate the effects of scanning along the ring.

Figure 21 shows the performance results. As shown, there is practically no overhead to using *scanRange* as compared with the application level search; again, this indicates that the price of consistency is low. To our surprise, the time needed to complete the range search, for either approach, does not increase significantly with the increased number of hops. On further investigation, we determined that this was due to our experiments running on a cluster in the local area network. In a wide area network, we expect the time to complete a range search to increase significantly with the number of hops.

6.3.3 *Evaluating* leave *and* Replicate to additional hop

In this section, we investigate the overhead of the proposed *leave* and *replicate to additional hop* operations as compared to the naive approach of simply leaving the ring without contacting any peer. For this experiment, we start with a system of 30 peers and delete

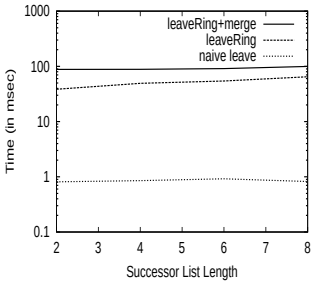

Figure 22. Overhead of leave

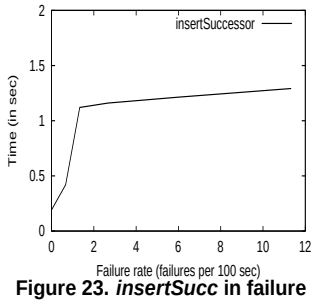

Figure 23. *insertSucc* in failure mode

items from the system that cause peers to merge and leave the ring.

We measure the time elapsed for three operations: (1) the *leave* operation in the ring, and (2) the merge operation in the Data Store (which includes the time for *replicate to additional hop*), and (3) the naive *leave*. Figure 22 shows the variation of the three times with successor list length. Note the log scale on y-axis. We observe that the *leave* and merge operations take approximately 100 msec, and do not constitute a big overhead. The naive version takes only 1 msec since it simply leaves the system.

6.3.4 *Evaluation in Failure Mode*

We have so far studied the overhead of our proposed techniques in a system without failures. We now look at how our system behaves in a system with failures. In particular, we measure the variation of the average time taken for an insertSucc operation with the failure rate of peers. The system setting is as follows: We insert one peer every three seconds into the system, and we insert two items every second. We use the default successor list length (4) and default ring stabilization period (4 sec).

Figure 23 shows the variation of average time taken for a insertSucc operation with the peer failure rate. We observe that even in the case when the failure rate is as high as 1 in every 10 seconds, the time taken for insertSucc is not prohibitive (about 1.2 seconds compared to 0.2 seconds in a stable system).

7. RELATED WORK

There has been a flurry of recent activity on developing indices for structured P2P systems. Some of these indices can efficiently support equality queries (e.g., [28, 32, 29]), while others can support both equality and range queries (e.g., [1, 2, 5, 6, 10, 12, 14, 15, 31]. This paper addresses query correctness and availability issues for such indices, which have not been previously addressed for range queries. Besides structured P2P indices, there are unstructured P2P indices such as [8, 13]. Unstructured indices are robust to failures, but do not provide guarantees on query correctness and

item availability. Since one of our main goals was to study correctness and availability issues, we focus on structured P2P indices.

There is a rich body of work on developing distributed index structures for databases (e.g., [18, 19, 21, 23, 24]. However, most of these techniques maintain consistency among the distributed replicas by using a *primary copy*, which creates both scalability and availability problems when dealing with thousands of peers. Some index structures, however, do maintain replicas lazily (e.g., [19, 21, 24]). However, these schemes are not designed to work in the presence of peer failures, dynamic item replication and reorganization, which makes them inadequate in a P2P setting. In contrast, our techniques are designed to handle peer failures while still providing correctness and availability guarantees.

Besides indexing, there is also some recent work on other data management issues in P2P systems such as complex queries [11, 16, 26, 27, 33, 34]. A correctness condition for processing aggregate queries in a dynamic network was proposed in [3]. An interesting direction for future work is to extend our techniques for query correctness and system availability to work for other complex queries such as keyword searches and joins.

8. CONCLUSION

We have introduced the first set of techniques that provably guarantee query correctness and system and item availability for range index structures in P2P systems. Our techniques provide provable guarantees, and they allow applications to abstract away all possible concurrency and availability issues. We have implemented our techniques in a real distributed P2P system, and quantified their performance.

As a next step, we would like to extend our approach to handle more complex queries such as joins and keyword searches.

9. ACKNOWLEDGEMENTS

This work was supported by NSF Grants CRCD-0203449, ITR-0205452, IIS- 0330201, and by AFOSR MURI Grant F49620-02-1-0233. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

10. REFERENCES

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

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

[3] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD*, 2004.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, 1987.

[5] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *Proc. SIGCOMM*, 2004.

[6] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. P-ring: An index structure for peer-to-peer systems. In *Cornell Technical Report*, 2004.

[7] A. Crainiceanu, P. Linga, A. Machanavajjhala, J. Gehrke, and J. Shanmugasundaram. A storage and indexing framework for p2p systems. In *WWW Poster*, 2004.

[8] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer networks. In *ICDCS*, 2002.

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

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

[11] L. Galanis, Y. Wang, S. Jeffery, and D. DeWitt. Locating data sources in large distributed systems. In *VLDB*, 2003.

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

[13] Gnutella - http://gnutella.wego.com.

[14] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in dynamic structured p2p systems. In *INFOCOM*, 2004.

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

[16] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *VLDB*, 2003.

[17] Jbi home page - http://www.rl.af.mil/programs/jbi/.

[18] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *IPPS*, 1992.

[19] T. Johnson and P. Krishna. Lazy updates for distributed search structure. In *SIGMOD*, 1993.

[20] D. Kossmann. The state of the art in distributed query processing. In *ACM Computing Surveys*, Sep 2000.

[21] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *SIGMOD*, 1994.

[22] P. Linga, A. Crainiceanu, J. Gehrke, and J. Shanmugasundaram. Guaranteeing correctness and availability in p2p range indices. In *Cornell Technical Report*, 2005.

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

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

[25] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1997.

[26] W. Ng, B. Ooi, K. Tan, and A. Zhou. Peerdb: A p2p-based system for distributed data sharing. In *ICDE*, 2003.

[27] V. Papadimos, D. Maier, and K. Tufte. Distributed query processing and catalogs for peer-to-peer systems. In *CIDR*, 2003.

[28] S. Ratnasamy, M. H. P. Francis, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

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

[30] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[31] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A peer-to-peer framework for caching range queries. In *ICDE*, 2004.

[32] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[33] I. Tatarinov and A. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD*, 2004.

[34] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Towards high performance peer-to-peer content and resource sharing systems. In *CIDR*, 2003.