An Indexing Framework for Structured P2P Systems

Adina Crainiceanu   Prakash Linga   Ashwin Machanavajjhala
Johannes Gehrke   Carl Lagoze   Jayavel Shanmugasundaram

Department of Computer Science, Cornell University
{adina,linga,mvnak,johannes,lagoze,jai}@cs.cornell.edu

Abstract

Different collaborative applications in a peer-to-peer (P2P) environment impose different requirements on the underlying P2P system. In this paper, we present a modularized indexing framework that cleanly separates the functional components of a structured P2P index, thereby allowing an application to tailor an index to its needs, while reusing components developed in previous systems. Our main contribution is a systematic way of handling query correctness, concurrency, and failures in the context of our indexing framework. Our techniques provide provable correctness and availability guarantees in a dynamic P2P system. In a simulation study, we use our indexing framework to compare the performance of three different P2P indices proposed in the literature.

1 Introduction

There are many applications that benefit from the cooperation of multiple peers. These applications range from simple file sharing to robust Internet-based storage management to digital libraries. Each of these applications impose different requirements on the underlying P2P infrastructure. For example, file-sharing applications require equality search and keyword search capabilities, but do not need sophisticated fault-tolerance. On the other hand, storage management requires only simple querying, but requires robust fault-tolerant properties. Digital library applications require both complex queries, including equality, keyword search, and range queries, and sophisticated fault-tolerance. Other applications such as service discovery on the Grid and data management applications impose their own requirements on the underlying P2P system.

One solution to this problem is to devise a special-purpose P2P infrastructure for each application. Clearly, this is quite wasteful, and does not leverage the common capabilities required across many (but not all) applications. We thus propose a modularized P2P indexing framework that cleanly separates different functional components. Our indexing framework is designed for structured P2P systems [3], which impose structure on the underlying dynamic P2P network in order to provide guarantees on load-balancing, search and availability.

We believe that our indexing framework offers three main benefits. First, as described above, it allows applications to tailor the system to their needs. Second, it allows reusing existing algorithms for different components rather than implementing everything anew. Finally, our framework allows experimenting with different implementations for the same component so that the benefits of a particular implementation can be clearly evaluated and quantified. This is especially important in an emerging field like P2P databases, where there are no universally agreed upon requirements or indices. We illustrate the power of this approach by instantiating three existing P2P indices: Chord [26] (which supports equality queries), Skip-Graphs [2], and P-Ring [4] (which supports both equality and range queries) in the context of our framework.

The main contribution of this paper is a systematic way of handling query correctness, concurrency, and failures in a dynamic P2P environment. We first show how existing P2P indices such as Chord [26] can have subtle concurrent interactions that can cause query results to be missed in some cases, even in an operational system. Further, these indices (originally developed for equality queries) also suffer from reduced availability when directly adapted for range queries. One solution to this problem is to simply let the application handle the query correctness and availability issues – this, for instance, is the approach taken by CFS [7] and PAST [25], which are applications built on top of Chord and Pastry, respectively. However, this approach exposes low-level concurrency details to applications and is also very error-prone due to subtle concurrent interactions between system components. In contrast, our approach is to embed novel techniques for ensuring query correctness and fault-tolerance directly into our indexing 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 correct-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 2005 CIDR Conference

ness 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 environment.

The second contribution of this paper is an implementation of three P2P indices, Chord [26], SkipGraphs [2] and P-Ring [4], in the context of our framework. In a simulation study, we compare the performance of the three indices and study the component-wise breakdown of the overall system cost. This is a first step towards a larger ongoing study of performing an "apples-to-apples" comparison of different P2P indices by instantiating them in a common framework.

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

2.2 Architecture

A P2P index needs to reliably support the following operations: search, item insertion, item deletion, peers joining, and peers leaving the system. We now introduce our modularized indexing framework, which is designed to capture most structured P2P indices. The main challenges in designing this framework are defining the relevant functional components and defining a simple API for each component so that the overall system is flexible enough to capture most of the existing structured P2P indices, while extensible enough so new P2P indices can be developed based on this framework. Figure 1 shows the components of our framework. These components and their interactions are described next. Note that the architecture does not specify *implementations* for these components but only specifies

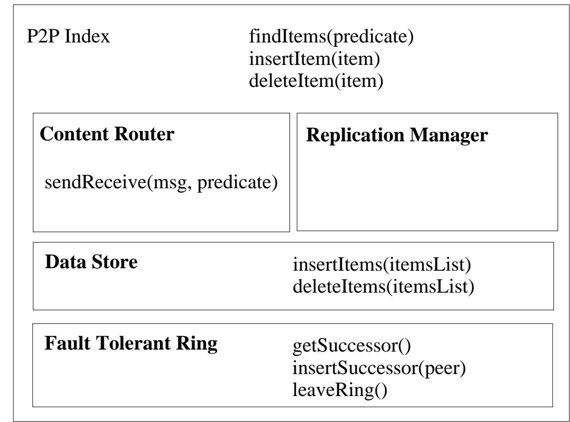

Figure 1: Indexing Framework

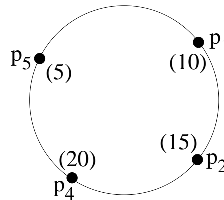

Figure 2: Ring

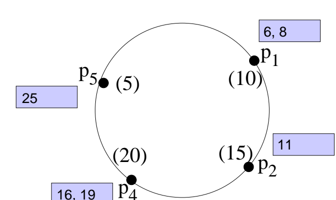

Figure 3: Data Store

functional requirements.

Fault Tolerant Hypersphere. The Fault Tolerant Hypersphere connects the peers in the system on a hypersphere, 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 hypersphere). On a ring, for a peer p, we can define $\mathtt{succ}(p)$ (respectively, $\mathtt{pred}(p)$) 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 $\mathtt{succ}(p_5) = p_2$, 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 thus increases clockwise around the ring (wrapping around at the highest value). The values of the peers in Figure 2 are shown in brackets.

Figure 1 shows the ring API. When invoked on a peer p, `p.getSuccessor` returns the address of $\mathtt{succ}(p)$. `p.insertSuccessor(p')` makes p' the successor of p when p (or p') wishes to join an existing ring. `p.leaveRing` 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).

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 $\texttt{pred}(p).val$ and $p.val$. We refer to the range $(\texttt{pred}(p).val, p.val]$ as $p.range$. Figure 3 shows an example Data Store that maps some search key values to peers on the ring. For example, peer p_3 is responsible for search key values 16 and 18. 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 [26], splitting and/or merging [10, 4]) 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.

Replication Manager. The Replication Manager is responsible for reliably storing items in the system even in the presence of failures, until items are explictly 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, then 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 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.

2.3 Benefits of the Indexing Framework

We believe that our indexing framework offers three primary benefits. First, it provides a principled way to implement and compare against existing indexing structures. For instance, Chord [26], which is a fault-tolerant P2P index that supports equality queries, can be implemented in our framework as follows. The Fault-Tolerant Ring is implemented using Chord's ring, and each peer is assigned a value using consistent hashing. The Data Store is implemented using a scheme that hashes each search key value to a value on the ring, and stores the corresponding item to the peer responsible for the range containing that value. The Replication Manager is instantiated using the techniques proposed in CFS [7]. The Content Router is implemented using Chord's finger tables. Other structures proposed in the literature such as CAN [23], Pastry [24], and SkipGraphs [2], can similarly be instantiated in our framework.

The second benefit of our framework is that it allows us to develop new P2P indices by leveraging parts of existing structures. In particular, we have recently developed a new P2P index called P-Ring [4] for equality and range queries, that reuses the Chord Fault Tolerant Ring

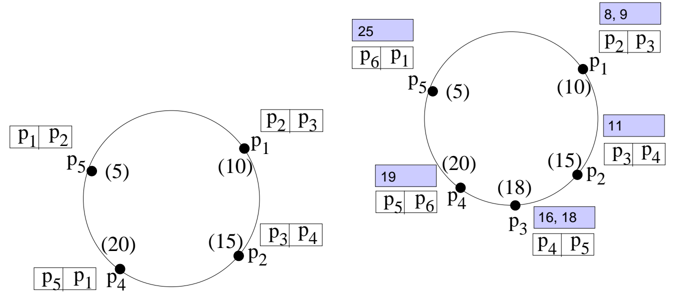

Figure 4: Chord Ring

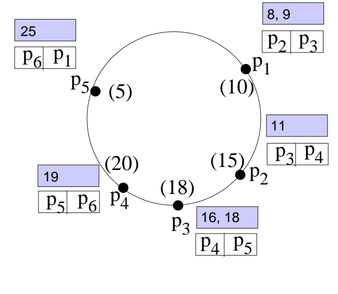

Figure 5: PRing Data Store

and Replication Manager, but develops a new Data Store and Content Router.

Finally, our framework allows extending the functionality of existing systems. For instance, Skip Graphs [2] is a P2P index that supports range queries but only supports one item per peer. We implemented the SkipGraphs Content Router on top of the P-Ring Data Store, thereby allowing Skip Graphs to handle multiple items per peer. We evaluate this implementation of Skip Graphs in Section 6.

2.4 An Example Instantiation: P-Ring

We now discuss the instantiation of P-Ring, a novel index structure designed for *range queries* in addition to equality queries. While the full details of P-Ring are left to another paper [4], we focus on the instantiation of P-Ring in the context of our framework. We will use this instantiation as a running example to illustrate the query correctness, concurrency and availability issues in subsequent sections. As mentioned earlier, in P-Ring, we reuse the Fault Tolerant Ring of Chord and the Replication Manager of CFS, and only devise a new Data Store and a Content Router for handling data skew in range queries.

Fault Tolerant Ring. P-Ring uses the Chord ring to maintain connectivity. The Chord ring achieves fault-tolerance by storing a *list* of successors at each peer, instead of storing just one successor. Thus, even if the successor of a peer p fails, p can use its successor list to identify other peers so that ring connectivity can be maintained. Figure 4 shows an example Chord ring in which successor lists are of length 2 (i.e., each peer p stores $\texttt{succ}(p)$ and $\texttt{succ}(\texttt{succ}(p))$ 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.

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 [23, 24, 26]. 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

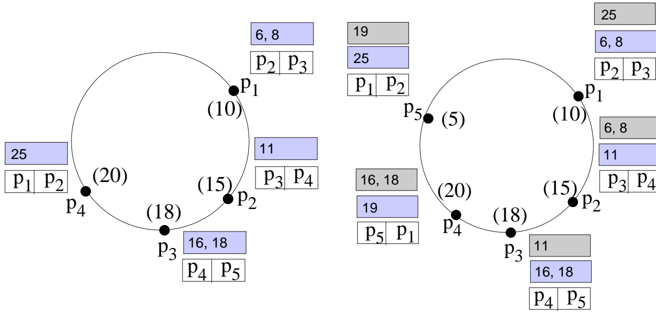

Figure 6: Data Store Merge   Figure 7: CFS Replication

cannot be used to handle range queries efficiently).

To solve this problem, the P-Ring Data Store assigns data items to peers 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. We thus need a way to dynamically reassign and maintain the ranges associated to the peers. The P-Ring Data Store achieves this goal by introducing two operations, `split` and `merge`, for handling item overflows and underflows in peers as described below.

The P-Ring Data Store has two types of peers: *live* peers and *free* peers. Live peers are 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 maintains the number of items stored in each live peer between $\ell = \texttt{sf}$ and $u = 2 \cdot \texttt{sf}$, where $\texttt{sf}$ is the "storage factor". Whenever the number of items in a peer p's Data Store becomes larger than $u = 2 \cdot \texttt{sf}$ (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 free peer. Whenever the number of entries in p's Data Store becomes smaller than $\ell = \texttt{sf}$ (due to deletions from $p.range$), we say that an *underflow* occurred. In this case, p tries to acquire a larger range and more entries 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 free peer.

As an illustration of a split, consider the Data Store shown in Figure 3. Assume that $\texttt{sf}$ 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.

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

Replication Manager. P-Ring uses the CFS Replication Manager, which works as follows. Consider an item i stored in the Data Store at peer p. The Replication Manager also replicates i to the k successors of p. In this way, even if p fails, i can be recovered from one of the successors of i. 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 this data structure are not relevant for this paper; see [4] for details.

3 Correctness and Availability: Goals

In instantiating indices in our indexing framework, perhaps the most interesting and challenging issue we faced was ensuring query correctness and system availability in the presence of concurrent peer insertions, deletions and failures. Specifically, we had the following design goals (we state these goals more formally in later sections):

- **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 and merges).
- **Item Availability:** The availability of items in the index should not be reduced due to index maintenance operations (such as splits and merges).

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 [7] and PAST [25], which are applications built on top of Chord and Pastry, respectively. 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 differ-

ent layers of the system. Specifically, we embed some 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.

We note that our proposed techniques differ from distributed database techniques [17] 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 themselves are not known a priori and can change *during* query processing).

In the subsequent two sections, we describe our solutions to query correctness and system/item availability. In the discussion, we use P-Ring as our running example and illustrate the issues and solutions in the context of the Chord Fault Tolerant Ring, the P-Ring Data Store, and the CFS Replication Manager. We note, however, that our solutions are more generally applicable and can be implemented in any instantiation that respects the API of our indexing framework.

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 our 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.

4.1 Defining Correct Query Results

Intuitively, a system returns a correct result for a query Q iff 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, be inserted, and items can move between peers during the duration of a query. To capture this, we first introduce the notion of a live item at a given time instant in the context of our indexing framework.

Definition (Live Item): An item i is *live* at time t in index I, denoted by $live_I(i,t)$ iff there exists a peer p in I at time t such that p's Data Store contains item i at time t and $\mathcal{M}(i.skv) \in range_{p,t}$, where $range_{p,t}$ is the value of $p.range$ at time t.

In other words, an item i is live at time t iff some peer with the appropriate range contains i in its Data Store at time t. 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 predicate.

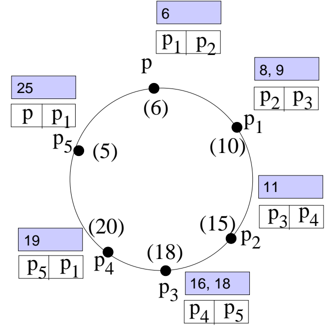

Figure 8: Peer p just inserted into the system

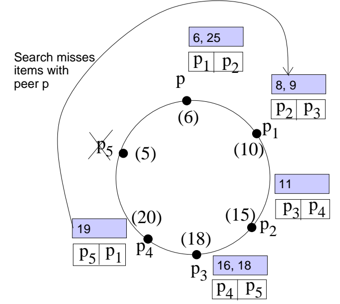

Figure 9: Incorrect query results: Search Q originating at peer p_4 misses items in p

Definition (Correct Query Result): A set R is a *correct query result* for a query Q initiated at time t_{begin} and successfully completed at time t_{end} in index I iff the following two conditions hold:

1. $\forall i \in R(satisfies_Q(i) \wedge \exists t(t_{begin} \leq t \leq t_{end} \wedge live_I(i,t)))$
2. $\forall i(satisfies_Q(i) \wedge \forall t(t_{begin} \leq t \leq t_{end} \Rightarrow live_I(i,t))) \Rightarrow i \in R$.

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

Evaluating a range query involves 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 the P-Ring Content Router, and the related concurrency issues have been described and solved elsewhere in the literature [4]. We thus focus on the second step and show how using the regular Chord Fault Tolerant Ring 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, the loose coupling between the ring and higher layers 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 such that $\mathcal{M}(i.skv) = 6$ 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 stablization 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 predecessorr, and it should accept the message. Again, the basic problem is that a peer does not know precise information about other peers in the system (due to the dynamics of a 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 Loose Coupling Between Layers We now illustrate how loose coupling between the ring and higher layers can produce inconsistent query results, *even if the ring is fully consistent*. Consider again the system in Figure 5 and assume that a query Q with query range $(20, 10]$ is issued at p_5. Since the lower bound of $p_5.range$ is the same as the lower bound of the query range, the sequential scan for the query range starts at p_5. The sequential scan operation first gets the data items in p_5's Data Store, and then gets the successor of p_5 along the ring (which, at this time, is p_1). Now assume that, as in the earlier example, an item i such that $\mathcal{M}(i.skv) = 6$ is inserted into the system. As before, p_1 splits and a new peer p is added as p_5's successor, as shown in Figure 8.

Now assume that the sequential scan of the query resumes, and the scan operation will propagate the scan to p_1 (the successor of p_5 at the time it got the successor). Again, the items in p will be missed, even though the items in p's range were live throughout the query execution. This problem arises because of the loose coupling between the ring and sequential scan operation - between the time the sequential scan operation got the successor of p_5 and the time it actually contacted this successor, the successor of p_5 had changed. Note that this problem can occur even when the ring is fully consistent (in our example, p_5's successor pointers after the insertion of p are in fact fully consistent).

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 Fault Tolerant Ring – higher layers can be guaranteed correctness by accessing the ring functionality through the appropriate API. We first present a solution that addresses ring inconsistency, and then present a solution that addresses loose coupling.

4.4 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 `insertSuccessor` (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 initally 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 `insertSuccessor` that satisfies this property for `JOINED` peers.

4.4.1 Defining Consistent Successor Pointers

We first introduce some notation. $p.succList_t$ is the successor list of peer p at time t. $succList.length$ is the length (number of pointers) of $succList$, and $succList[i]$ ($0 \leq i < succList.length$) refers to the i'th pointer in $succList$. The value of a peer at time t is $p.val_t$. A peer is said to be non-failed at time t if it did not fail at any time $t' \leq t$.

Definition (Consistent Successor Pointers): A set of non-failed peers $\mathcal{P}$ at time t has *consistent successor pointers at time* t iff the following condition holds:

- $\forall p \in \mathcal{P}, \forall i \ (0 \leq i \leq p.succList_t.length \wedge p.succList_t[i] \in \mathcal{P}) \Rightarrow \forall p_1 \in \mathcal{P} \ (p_1.val_t \in (p.val_t, p.succList_t[i].val_t) \Rightarrow \exists j (0 \leq j < i \wedge p.succList_t[j] = p_1))$

In other words, the successor pointers of a set of peers $\mathcal{P}$ is consistent iff for every peer $p \in \mathcal{P}$, if p has a pointer to a peer $p' \in \mathcal{P}$ in its successor list, it also has pointers to all peers $p_1 \in \mathcal{P}$ such that $p_1.val \in (p.val, p'val)$. Intuitively, this means that p cannot have "missing" pointers to peers in the set P. 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_1 but not to p.

4.4.2 Proposed Algorithm

We first present the intuition behind our algorithm, before presenting the details. Assume that a peer p' is to be inserted as a successor of a peer p. Intially, 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 exists some JOINED peers p_x and p_y such that: $p_x.succList[i] = p$ and $p_x.succList[i + k] = p_y$ (for some $k > 0$) and for all $j, 0 < j < k$, $p_x.succList[i + j] \neq p'$. In other words, p_x has a pointer to p and p_y but not a pointer to p' whose value occurs between $p.val$ and $p_y.val$.

Our algorithm avoids this case by ensuring that at the time of transitioning p' 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. This propagation of p' pointer is piggybacked on the Chord ring stabilization protocol, and hence does not introduce new messages.

Algorithms 1 and 2 show the pseudocode for the insertSuccessor method and the modified ring stabilization protocol, respectively. In the algorithms, we assume that in addition to $succList$, each peer also has a list called $stateList$ which stores the state (JOINING or JOINED) of the corresponding peer in $succList$. We now 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 insertSuccessor 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 $stateList$ (lines 5) and blocks waiting for an acknowledgment method 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 10 (JOINING list entries are marked with a "*").

Algorithm 1 : p_1.insertSuccessor(*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 ack from some predecessor; 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
```

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 Figure 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

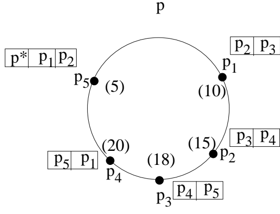

Figure 10: After p_5.insertSuccessor call

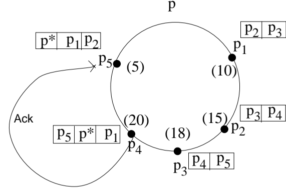

Figure 11: Propagation and final ack

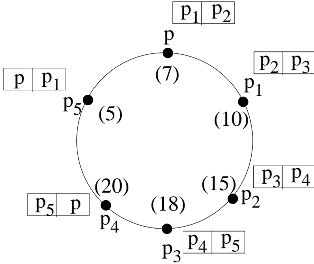

Figure 12: Completed `insertSuccessor`

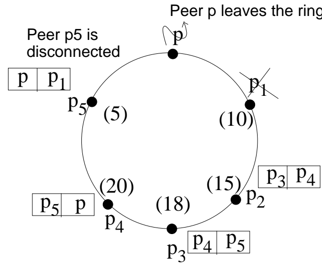

Figure 13: Naive merge leads to decreased reliability

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 11.

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 (p_5) in the predecessor list 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 `insertSuccessor` 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 occured is shown in Figure 12. We can prove the following theorem (the proof can be found in [5]):

Theorem 1 *If P_t is the set of peers in the* `JOINED` *state at time t, then Algorithms 1 and 2 ensure that P_t has consistent successor pointers at time t.*

In addition, we can prove the following theorem about the liveness of the `insertSuccessor` operation (i.e., that it will complete in a bounded time) given a certain peer failure rate, even in the presence of adversarial failures. We use the notation $p.state_t$ to denote the state of peer p at time t.

Algorithm 3 : $p.sendToSuccessor(msg)$

1: readLock $succList$
2: send msg to first live successor, and get response $retMsg$
3: releaseLock $succList$
4: return $retMsg$

Algorithm 4 : $p.sendToSuccessorHandler(msg)$

1: readLock $succList$
2: $retMsg = upCall(msg)$
3: releaseLock $succList$
4: return $retMsg$

Theorem 2 *If the stabilization procedure shown in Algorithm 2 runs at all non-failed peers at least once every T time units, and at most one peer fails every F time units, and $T < F$, then there exists a constant c bounded by $O(TF/(F-T))$ such that if the* `insertSuccessor(p')` *method is invoked on peer p at time t, and if both p and p' are non-failed at time $t+c$, then $p'.state_{t+c} =$* `JOINED`.

We can also prove a stronger statement if we assume uniform (as opposed to adversarial) peer failures [5].

4.5 Handling Loose Coupling

Recall from the discussion in Section 4.2.2 that even if the ring is fully consistent, query results can be missed due to the loose coupling between the ring and the higher layers. Essentially, the problem is that the successor of a peer p can change in the ring while higher layers are trying to contact an old successor of p. How do we shield the higher layers from the concurrency details of the ring while still ensuring correct query results?

Our solution to this problem is as follows. We introduce a new API method for peer p on the ring called `sendToSuccessor`, which sends a message from a higher layer (such as the range scan operation) to the current successor of p in the ring. Since `sendToSuccessor` handles all the concurrency issues associated with the ring, higher layers do not have to worry about successor pointers changing. Algorithm 3 shows the pseudocode for the `sendToSuccessor` method in a peer p. The method first acquires a read lock on $succList$ (to freeze the ring) and then sends a message to the first non-failed peer in $succList$ (lines 1-2). On the receiving end, the peer p' receiving the `sendToSuccessor` message runs the algorithm shown in Algorithm 4. p' first acquires a read lock on its $succList$ (to freeze its ring), then throws up the message to the higher layer, gets the higher layer reponse and returns the response (lines 1-4). The `sendToSuccessor` method at p then releases the lock on its $succList$ and returns the response to the higher layer. We can prove the following theorem about the correctness of `sendToSuccessor`.

Theorem 3 *If a peer p' receives a* `sendToSuccessor` *message from peer p at time t and if $\mathcal{P}_t$ is the set*

Algorithm 5 : $p.scanAlongRing($`Query Q`$)$

1: **if** $\texttt{succ}(p_1).range \cap Q \neq \phi$ **then**
2: // send a search message with query Q
3: $results = sendToSuccessor(searchMsg)$
4: **end if**
5: $results \mathrel{+}= \{\, i | i.skv \in p.range \wedge i.skv \in Q \,\}$
6: return $results$

of JOINED *peers at time* t*, then* $p' \in \mathcal{P}_t \wedge \forall p_x \in \mathcal{P}_t(p_x.val_t \notin (p.val_t, p'.val_t))$

Using the `sendToSuccessor` method, we can ensure correct query results for range scans as follows. For a query Q with query range $(l, u]$, we assume that the range scan is initiated in peer p at time t such that $p.state_t = \texttt{JOINED} \wedge p.val_t \in (l, u) \wedge \forall p'(p'.state_t = \texttt{JOINED} \Rightarrow p'.val_t \notin (l, p.val_t))$ (i.e., the range scan is initated on the first peer p in the ring that overlaps with the query range, due to the correctness of the Content Router). The pseudocode for the scan operation at p is shown in Algorithm 5. It first checks to see if the scan should be forwarded to $\texttt{succ}(p)$ based on the query range; if so, it invokes the `sendToSuccessor` method with a $searchMsg$ and gets the results (lines $2-5$). p then adds the items from its own Data Store that satisfy the query range to the results, and returns these results. The `sendToSuccessor` handler for the $searchMsg$ runs the same pseudocode as Algorithm 5.

It is easy to see why the inconsistency described in Section 4.2.2 does not occur with this method. We can prove:

Theorem 4 *For a query* Q *with query range* $(l, u]$*, if the range scan algorithm in Algorithm 5 is initiated in peer* p *at time* t *such that* $p.state_t = \texttt{JOINED} \wedge p.val_t \in (l, u) \wedge \forall p'(p'.state_t = \texttt{JOINED} \Rightarrow p'.val_t \notin (l, p.val_t))$*, then the algorithm produces* correct *query results (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 and merges). Similarly, ensuring item availability means that the availability of items should not be reduced due to index 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. We refer the reader to [5] for the details.

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 [26] when the only operations on the ring are peer insertions (splits) and failures. These availability guarantees also carry over to our variant of the Fault Tolerant Ring with the new implementation of `insertSuccessor` described earlier because it

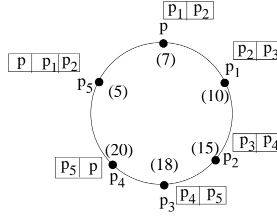

Figure 14: Controlled leave of peer p

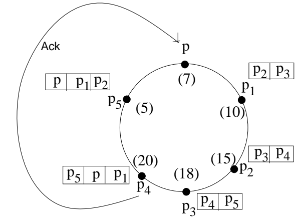

Figure 15: Final ack received at peer p. Peer p is good to go.

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 `leaveRing` operation in the Fault Tolerant Ring.

We now show that a naive implementation of `leaveRing`, which is simply removing the merged peer from the ring, does in fact reduce system availability. We then sketch an alternative implementation for the `leaveRing` 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 `leaveRing` Reduces System Availability: Consider the system in Figure 12 in which the length of the successor list of each peer is 2. Without `leaveRing`, 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 during a failure). We now show that in the presence of the naive `leaveRing`, a single failure can disconnect the ring. Thus, `leaveRing` reduces the availability of the system. The example is as follows. Assume that `leaveRing` 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 13, 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 `leaveRing` 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 `insertSuccessor` case, we piggy-back the lengthening of the successor lists on the ring stabilization protocol. This is illustrated in the following example.

Consider Figure 12 in which `leaveRing` is invoked on p. During the next ring stabilization, the predecessor of p - p_5 - increases its successor list length by 1. The state of the system is shown in Figure 14. During the next

ring stablization, the predecessor of p_5 - 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 15. It is easy to see that if p leaves the ring at this point, a single failure cannot disconnect the ring as in the previous case. We can formally prove that the new algorithm for `leaveRing` does not reduce the availability of the system [5].

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 time t into an index I by $insert_{I,t}(i)$, and the deletion of an item i' at time t' as $delete_{I,t'}(i')$.

Definition (Item Availability): An index I is said to preserve *item availability* with respect to a set of insertions Ins and a set of deletions Del iff $\forall i \forall t (\exists t'(t' \leq t \wedge insert_{I,t'}(i) \in Ins \wedge \forall t''(t' \leq t'' \leq t \Rightarrow delete_{I,t''}(i) \notin Del))) \Rightarrow live_I(i,t)$.

In other words, for all time t when an item i has been inserted into the system and not deleted, i is a live item.

The CFS Replication Manager, implemented on top of the Chord Ring provides strong guarantees [7] 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 could compromise item availability is the `leaveRing` (merge) operation. We now show that using the original CFS Replication Manager in the presence of merges does in fact 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. Here, the top box associated with each peer represents the items replicated at that peer (recall that 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 does a `leaveRing` 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 16. If p_5 fails at this time (this is the single failure), the items 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 and Replication Manager for one additional hop before p leaves the system. This is illustrated in Figure 17, where before p_1 merges with p_2, it increase the replicas for items in its Data Store and Replication Manager by one additional hop. Then, 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 again prove that the above scheme preserves item availability even in the presence of concurrent splits and merges [5].

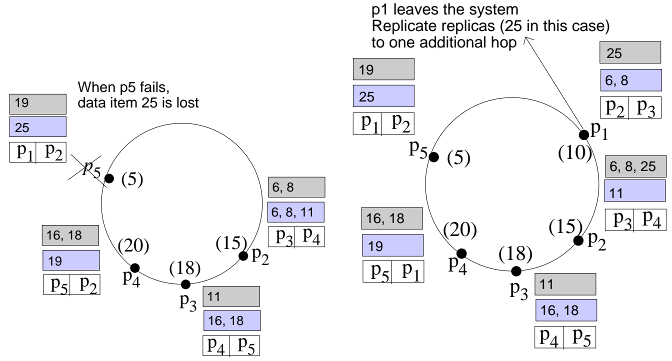

Figure 16: Peer p_5 fails causing loss of item 25

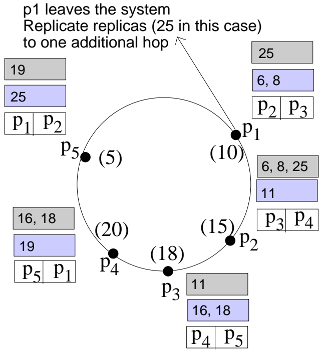

Figure 17: Replicate item 25 one additional hop.

6 Experimental Evaluation

We had two main goals in our experimental evaluation: (1) to use our indexing framework to implement and compare different P2P indices proposed in the literature, and (2) to demonstrate the feasibility of our proposed query correctness and availability algorithms in a dynamic P2P system. Towards these goal, we implemented and compared the Chord, SkipGraphs and P-Ring indices using a simulator version of our indexing framework, which simulated concurrent peer execution and failures. We note that the results presented here are by no means exhaustive, but are initial steps towards a larger, more comprehensive study.

6.1 Experimental Setup

We implemented Chord, SkipGraphs and P-Ring in the simulator version of our indexing framework. In the experiments, we compare the performance of the following instantiations: Chord, SkipGraphs with alphabet size 2, SkipGraphs with alphabet size 10, P-Ring of order 2, and P-Ring of order 10. We chose 2 and 10 as the alphabet size/order because they illustrate some interesting tradeoffs in the system performance. The simulator 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.

We consider three main performance metrics. The first is the the *maintenance message cost*, which is the number of messages per peer per minute (or 60 simulation time units) required to maintain the index components. The second metric is the *maintenance bandwidth cost*, which is the same as the previous metric, except that it counts message size in bytes instead of number of messages. The final metric is the *search cost*, which is the number of messages required to evaluate a query. In our experiments, we calculate the search cost by averaging the number of messages required to search for a random value in the system starting from 100 random peers. Since the main variable

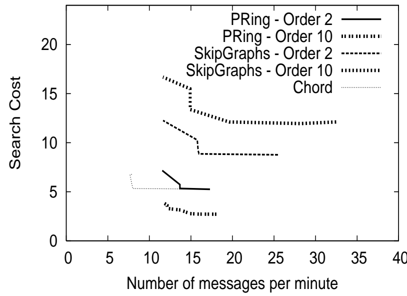

Figure 18: Search vs Message Cost

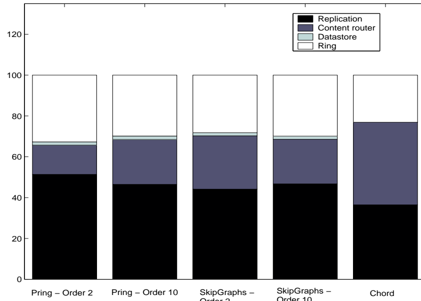

Figure 19: Cost per Component

component of the cost of range queries is finding the data item with the smallest qualifying value (retrieving the other values has a fixed cost of traversing the relevant leaf values), we only measure the cost of finding the first item for range queries. This also enables us to compare the performance of P-Ring and SkipGraphs, designed for range queries, against the performance of Chord, designed for equality queries.

We use a two-phase system evolution model in our experiments. In the first phase, peers are inserted until there are a total of 2000 peers in the system. In the second phase, the system is under "churn", whereby peers are inserted into and deleted (failed) from the system with equal probability. For a given rate of churn (i.e., rate of peer insertions/deletions), we run the different components at what we call their *survivability rate*. For the Fault-Tolerant Ring, this is the minimum stabilization rate so as to maintain connectivity of the ring. For the Replication Manager, this is the minimum replication rate so as to preserve item availability when peers fail at the churn rate. By running the system components at their survivability rates, we can identify the minimum cost for each component that still ensures system correctness.

6.2 Experimental Results

We present experimental results for a churn rate of 2 (i.e., two peers are inserted/failed every simulation time unit). Figure 18 shows the effect of peer insertions and failures on index performance. The basic tradeoff is between the search cost and the maintenance message cost. When the Content Router is stabilized at a high rate, the maintenance message 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 slowly, the maintenance message cost decreases but search cost increases.

As shown in Figure 18, P-Ring always dominates Skip Graphs due to its superior search performance; this is because, even in a stable system, the search performance of Skip Graphs with alphabet size of d is $O(d \times \log_d(N))$, where N is the number of peers in the system, as compared to the search cost of $O(\log_d(N))$ for a P-Ring of order d. Chord outperforms the P-Ring of order 2 because both have search cost $O(\log_2(N))$, but Chord does not have the overhead of dealing with splits and merges during system churn. However, P-Ring of order 10 offers a better search cost, albeit at a higher maintenance cost, while still supporting range queries. We also obtained similar results for search cost vs. maintenance bandwidth cost, and hence the results are not shown.

Figure 19 shows the component-wise breakdown of message costs for each index, when the maintenance message cost is fixed (we fixed this cost to be 17 in these experiments, but the results for other costs were similar). This breakdown illustrates how much each component contributes to the overall index maintenance cost. The results are quite interesting: Chord spends a significantly higher fraction of its messages stabilizing its Content Router than the other index structures; this is due to the fact that Chord performs a $O(\log(N))$ search during each stabilization, while the other indices only contact their successors at different levels in the index. However, the overhead of the Chord Data Store is relatively small because it does not have to split or merge for load balancing, while the overhead of the P-Ring Data Store is comparatively higher. In both cases, however, the fraction of messages in the Data Store is not very high because we only consider peer insertions/deletions, and not item insertions/deletions in this set of experiments. Finally, the graph shows that the overhead of reliable data storage (i.e., replication) is about 40% of the total message cost; this is due to the relatively high churn rate. Nevertheless, applications should make an informed decision as to whether this functionality is required because it comes at a fairly high price.

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., [23, 26, 24]), while others can support both equality and range queries (e.g., [1, 2, 4, 8, 10, 13, 12]. One of the contributions of this paper is a unified framework that can be used to implement and compare the above indices. In addition, 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 [11, 6]. Unstructured indices are very 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., [15, 16, 18, 19, 20]. 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., [16, 18, 20]). 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 indexing framework is 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 [9, 14, 21, 22, 27, 28]. An interesting direction for future work is to extend our techniques for query correctness and system availability to work for complex queries such as joins and aggregations.

8 Conclusion

We have presented a componentized indexing framework for structured P2P systems. The primary benefits of this framework are (a) applications can tailor an index to its needs, (b) it allows code reuse, and (c) it provides a systematic way of comparing different P2P indices. Our main focus was addressing query correctness and system/item availability issues in the presence of concurrent peer operations and failures, and implementing solutions for these in the context of our framework. In a simulation study, we used our framework to compare different P2P index structures. This work was done as part of the PEPPER project at Cornell, where the goal is to build a full-fledged P2P database system.

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] H. Balakrishnan, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. In *CACM*, 2003.

[4] 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.

[5] A. Crainiceanu, P. Linga, A. Machanvajjhala, J. Gehrke, and J. Shanmugasundaram. An indexing framework for structured p2p systems. In *Cornell Technical Report*, 2004.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] 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.

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

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