Guaranteeing Correctness of Lock-Free Range Queries over P2P Data*

Stacy Patterson, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science, University of California Santa Barbara
Santa Barbara, CA 93106, USA
{sep,agrawal,amr}@cs.ucsb.edu

Abstract. As P2P systems evolve into a platform for full-fledged distributed database management systems, the need arises for sophisticated query support and guarantees on query correctness. While there has been recent work addressing range queries in P2P systems, the work on query correctness is just beginning. Linga et al.[1] provided the first formal definition of correctness for range queries in P2P systems and described a lock-based range query technique that is provably correct. A natural question that arises is whether it is possible to develop a lock-free protocol that can meet the same guarantee of correctness. In this paper, we demonstrate the feasibility of lock-free correct protocols by first developing a simple, proof-of-concept query protocol and verifying that this protocol meets the correctness conditions. We then describe a more robust extended protocol and prove that for stable systems with only item insertions, item deletions, and item redistributions, this extension insures that every range query can be satisfied correctly.

1 Introduction

P2P systems provide the benefits of fault tolerance, load balancing, and scalability, making them a promising platform for distributed storage systems. Initial work on P2P systems focused on the development of distributed hash tables, or P2P indexes [2,3,4,5]. These P2P indexes allow for the storage of key/value pairs and the ability to do exact match search for an item using the item's key. In order to realize the full potential of P2P systems for distributed data management, these systems must provide support for more sophisticated query predicates. Recent work in the area of P2P range indexes enables efficient single and multi-dimensional range queries [6,7,8,9,10]. However, these systems do not provide any guarantees on the correctness of query results. Specifically, they do not guarantee that the query results include all and exactly those items that satisfy the range query predicate in the presence of item insertions, item deletions, and range redistributions. Such correctness guarantees are a necessary step in the evolution of P2P systems into full-fledged distributed database management systems.

* This research was funded in part by NSF grants IIS 02-23022, CNF 04-23336, and INT 00-95527.

G. Moro et al. (Eds.): DBISP2P 2005/2006, LNCS 4125, pp. 123–134, 2007.

Linga et al.[1] are the first to address correctness of range queries. This work provides a formal definition of correctness for range queries and describes a technique for range queries that is provably correct. The technique relies on locking to insure that no data items that satisfy the query predicate will be omitted from the result. A natural question that arises is whether it is possible to develop a lock-free technique that meets the same definition of correctness. In this paper, we demonstrate that a lock-free provably correct query protocol is feasible. We develop a simple, lock-free protocol in the context of P-Ring[7], the same P2P range index used in the lock-based approach. Our protocol returns only correct query results and rejects any query that cannot be satisfied correctly under the current system conditions. We also develop an extended protocol that greatly decreases the number of queries that will be rejected by the system. Specifically, in a stable ring where no new peers join and no peers fail or leave, this extension insures that every range query can be processed correctly without being rejected.

The remainder of the paper is organized as follows. In Section 2, we present the system model and provide background on P-Ring. In Section 3, we formalize the notion of query correctness. In Section 4, we describe a simple, correct range query protocol, and in Section 5, we describe a more robust extension to that protocol. Finally, we conclude in Section 6.

2 Background

2.1 System Model

The model we adopt is a generalization of many existing P2P systems. The system consists of a collection of peers, P, where a peer is a single processor that contributes some amount of storage space to be used by the system. Each peer, $p \in P$, has a unique physical identifier, such as an IP address. Each peer also has a unique logical identifier, denoted $p.id$, which is a key chosen from a discrete key space. We assume the existence of an underlying network layer that allows a peer to communicate with another peer using a direct communication channel. All messages are delivered within some known, bounded time-delay and message ordering is preserved within any given channel.

The system allows nodes to join and leave at any time. We assume a fail-stop model for peer departures. The system provides a $lookup(key)$ operation which locates, with high probability, the peer that is the immediate successor of the key. The system also provides operations for item insertion and item deletion. Additionally, the system supports single dimensional range queries of the forms $[lb, ub]$, $(lb, ub]$, $[lb, ub)$, and (lb, ub) where lb is the lower bound and ub is the upper bound of the range predicate.

2.2 P-Ring

We describe our techniques in the context of P-Ring, a P2P index framework that supports both range queries and exact match queries. Our techniques can be generalized to other range indexes that support single dimension range queries [6,8].

The P-Ring architecture is divided into several layers, each encapsulating specific functionality of the system. The foundation of a P-Ring system is a Chord ring [4] which provides the connectivity in the system. The Data Store layer sits on top of the Chord ring and is responsible for data item storage and load balancing operations. P-Ring also provides a Replication Manager that is responsible for maintaining replicas to improve item availability in the presence of peer failures. Finally, the P-Ring Content Router provides a routing structure that enables efficient $lookup(key)$ operations. We encapsulate the functionality needed for correctness in the lower levels of the system. Our approach is therefore limited to changes in the Chord ring and Data Store layers.

Chord Ring. In the Chord ring, each peer, p, stores the identity (both physical and logical) of its predecessor, denoted $pred(p)$, and its successor, denoted $succ(p)$, in the ring. p also keeps a list of additional successors as a redundancy measure in case $succ(p)$ fails. Each peer is responsible for a portion of the key space which is the range $(pred(p).id, p.id]$. These values are the lower and upper bounds of the peer's range, which we also denote by $p.lb$ and $p.ub$ respectively. In a consistent ring $pred(p).ub = p.lb$ for all peers, p, in the system. We assume that when a new peer joins the ring, its predecessor and successor are notified and update their successor and predecessor pointers to reflect the existence of the new peer as part of the join process.

Data Store. One of the goals of a P2P storage system is to evenly distribute data items among all of the peers in the system. In many systems, load balancing is achieved through the use of consistent hashing [11]. For example, in CFS [12], peer IDs are generated using a hash function that insures with high probability that the IDs are uniformly distributed across the key space. Item IDs are also generated using a hash function, and items are stored at the peer whose ID immediately succeeds the item ID. This approach is effective in insuring that every peer is responsible for roughly the same number of items. However, a hash function does not generate IDs that preserve the order of the items.

In P2P range indexes, the item ID assignment policy must preserve item order so that range queries can be answered efficiently. To maintain item order, P-Ring uses the search key of the item as the item ID, denoted $i.skv$ where i is an item. With this policy, a range query of the form $[lb, ub]$ can be answered by first doing a $lookup(lb)$ to locate the peer responsible for the lower bound of the query and then traversing along the Chord ring following successor pointers until the peer responsible for ub is reached.

Since the P-Ring ID assignment scheme does not guarantee uniform item distribution across peers, it is possible for a peer to become heavily loaded if a particular range contains too many items. It is also possible for a peer to be underloaded if its range is less popular. To address this issue, P-Ring provides three explicit load balancing operations, *split*, *merge*, and *redistribute*. If a peer becomes overloaded, it invites a new peer to join the ring and divides its range and the corresponding data items with the new peer through a *split* operation. If a peer's storage space is underutilized, it informs its successor through a *merge*

operation. The successor, in a *redistribute* operation, either gives part of its range and associated items to its predecessor or gives up its entire range to its predecessor and leaves the ring. With these load-balancing operations, P-Ring can guarantee that the number of items stored at each peer is between *sf* and *2sf* for some storage factor *sf*.

3 Query Correctness

We adopt the definition of *correct query results* given in [1]. This definition depends on the notion of a *history*, which describes the operations of the system and a partial ordering upon them. The partial order, $\leq$, is a "happened before" relationship such that for any two operations o_1 and o_2, we say that o_1 happened before o_2 if o_1 completed before o_2 began. If it is not the case that o_1 happened before o_2, it is possible the operations executed in parallel. The formal definition of a history is as follows.

Definition 1 (History). *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.*

The definition of a correct query result also relies on the definition of a *truncated history*.

Definition 2 (Truncated History). *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 $\forall o_1, o_2 \in O_{\mathcal{H}_o}(o_1 \leq_{\mathcal{H}} o_2 \Rightarrow o_1 \leq_{\mathcal{H}_o} o_2)$.*

In other words, a truncated history is a history that contains all and only those operations that happened before a particular operation.

3.1 Correct Query Results

Intuitively, a correct query result will contain exactly those items in the P2P system that satisfy the range predicate. The dynamic nature of the system complicates the definition of what it means for an item to be "in the system". At a high level, an item is in the system, or *live*, if it has been inserted at some peer and not yet been deleted from any peer. We use the same terminology from [1] but adopt a slightly different definition of a live item. $insertItem(i)$ denotes the successful insertion of item i into the system. $deleteItem(i)$ denotes the successful deletion of the item i from the system. We use $items_{\mathcal{H}}(p)$ to denote the collection of items stored at a peer p.

Definition 3 (Live Item). *An item i is live in a history $\mathcal{H}$, denoted $live_{\mathcal{H}}(i)$, iff $(insertItem(i) \in \mathcal{H}) \wedge (deleteItem(i) \notin \mathcal{H})$.*

Additionally, in P2P systems, we must address the issue of node failures with regards to the items stored at a failed node. When a node fails, its successor in the Chord ring assumes responsibility for the node's range, but only once the successor becomes aware of the failure. We consider this detection of failure to

be an implicit delete of every item stored at the failed peer. So, it is only after the failure is detected that these items are no longer live.

We now state the definition of a correct query result from [1]. In the definition, $satisfies_{\mathcal{Q}}(i)$ denotes whether item i satisfies the range predicate of query $\mathcal{Q}$.

Definition 4 (Correct Query Result). *Given a history $\mathcal{H}$, 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 if an item i is included in the query result, then it was live at some time during the query. The second condition states that every item that was live for the entire duration of the query is included in the result.

The goal of this paper is to explore lock-free techniques for range queries that produce correct query results according to the above definition.

3.2 Incorrect Query Results: Examples

In [1], the authors present two examples of how a naive approach to range query execution that consists of simply traversing along the ring will give incorrect query results. We summarize these examples here. In Section 4.3 we show how our lock-free implementation disallows these incorrect scenarios.

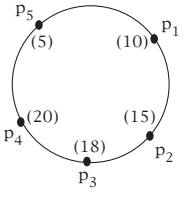

Fig. 1. Example P-Ring

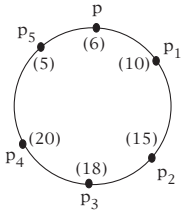

Fig. 2. P-Ring with new node

Inconsistent Successor Pointers. Consider the ring shown in Figure 1 and suppose each peer maintains a successor list of size 2. The successor list for p_4 is $\{p_5, p_1\}$. p_5 is responsible for the range $(20, 5]$ and p_1 is responsible for the range $(5, 10]$. Suppose that p_1 becomes overloaded and splits its range with a new peer p, as shown in Figure 2. When p joins, it becomes responsible for the range $(5, 6]$ and p_1 becomes responsible for the range $(6, 10]$. A query with range predicate $(20, 9]$ arrives at p_4, and p_4 responds to the query with the data items in the range $(20, 5]$. Then, p_5 fails. p_4 tries to forward the query to p_5 and detecting the failure, forwards the query to p_1. Note that p_4 has not yet updated its successor list to reflect the existence of p. p_1 will respond to the query with items in the range $(6, 9]$ and the range $(5, 6]$ will have been omitted from the query result.

Concurrent Redistribution. In the second scenario, we see that it is possible for a query to produce incorrect results even if the successor pointers are completely consistent. Consider again the ring in Figure 1. Suppose a query $(10, 18]$ arrives at p_2. p_2 returns the items in the range $(10, 15]$ and forwards the query to its successor, p_3. Suppose, at the same time, p_3 is in the process of redistributing part of its range with p_2 and has transferred the range $(15, 16]$ to p_2. When the query arrives at p_3, p_3 will return items in the range $(16, 18]$ and the items in $(15, 16]$ will be missing from the query result.

In both cases, the incorrect query results stem from the fact that there is some degree of uncertainty about the range for which a peer is responsible. If we can eliminate this uncertainty by clearly defining the range for which a peer can safely answer queries, then we can use this to produce a correct query protocol.

4 A Simple Protocol

First we examine how to remove any uncertainty in range responsibility that may be introduced through the *split*, *merge*, and *redistribute* operations. Then, we present a correct, lock-free range query protocol.

4.1 Range Ownership

When a peer performs a *split*, *merge*, or *redistribute* operation, its range changes. It is this change that is problematic for range queries. The first step to query correctness is to clarify range responsibility when these operations are performed. To do this, certain steps of each operation must be performed atomically. We outline these atomic steps for the *merge* operation. The atomicity requirements for *split* and *redistribute* are defined similarly.

When peer p experiences underflow, it invokes the *merge* operation to request that its successor relinquish some or all of its range to p. The result is that p increases the range it is responsible for and receives the data items associated with the range addition. The *merge* operation is given in Algorithm 1. In line 2, p alerts its successor of the underflow and waits for the successor's response. In line 4, p updates its set of items to include the items that were given up by its successor. In line 5, p updates its range to include the range relinquished

Algorithm 1. $p.merge()$

1. // send message to successor and wait for result
2. $(newRange, newItemList) = succ(p).initiateMerge(p, |p.range|)$;
3. // execute next two steps atomically
4. $p.list.add(newItemList)$;
5. $p.range.add(newRange)$;

by its successor. By executing steps 4 and 5 atomically, p insures that it will only process operations (item insertions, item deletions, lookups, and range queries) for the new range once it has incorporated all live items in the range into its Data Store.

4.2 Correct Range Queries

The simple query protocol is shown in Algorithm 2 and Algorithm 3. This is a slight modification of the original P-Ring protocol[1] without locking. The query begins with a *lookup*(lb) operation. *rangeQuery* is then invoked at the peer responsible for lb. *processQuery* is invoked at each subsequent peer that participates in the query ending at the peer responsible for ub. We assume for simplicity that *processQuery* and *rangeQuery* are each executed atomically. Rather than passing the same lower bound to its successor when forwarding the query along the ring, each peer p sends $p.ub$ as the lower bound. By doing so, p informs its successor of what part of the range query remains to be answered. The successor accepts the query only if it is able to exactly satisfy this lower bound, thus eliminating the possibility of gaps or duplicates in the query results.

We assume that during any query execution there are only a finite number of new peers entering the system. Therefore, every query terminates at some time.

Theorem 1. *Using the simple protocol, every query that is accepted (terminates with no peer rejecting it) produces a correct query result.*

Proof. Consider a query $Q = [lb, ub]$ that begins with operation o_s and ends successfully with operation o_e. The operations *rangeQuery* and *processQuery* insure that, if the query terminates without rejection, the intervals r satisfied at each of the peers that participate in the query are non-overlapping. The union of these intervals is exactly equal to $[lb, ub]$.

Since the union of the intervals satisfied at each peer equals $[lb, ub]$, $\forall i \in R\ satisfies_Q(i)$ holds. Let i be any item such that $i \in R$ and $satisfies_Q(i)$ holds. It must be the case that i was returned by some peer in line 6 of *processQuery* or *rangeQuery*. If we call this invocation of *processQuery* (or *rangeQuery*) operation o, then we have $insertItem(i) \leq_{\mathcal{H}} o$, and if i was deleted in this history, $o \leq_{\mathcal{H}} deleteItem(i)$. So, there exists an operation o such that, $o_s \leq o \leq o_e$ and also such that $live_{\mathcal{H}_o}(i)$. Therefore, Condition 1 of the the Correct Query Result definition holds.

Consider i such that $satisfies_Q(i)$ is true and $\forall o \in O_{\mathcal{H}}(o_s \leq_{\mathcal{H}} o \leq_{\mathcal{H}} o_e \wedge live_{\mathcal{H}_o}(i))$ holds. We claim that there must be some peer p such that $i \in p.range$ for the operation $p.processQuery$ (or $p.rangeQuery$). Suppose this is not the case. Then, for some peers p_1 and p_2 where p_2 is the successor of p_1, we have $i.skv > p_1.ub$ during the invocation of $p_1.processQuery$ (or $p_1.rangeQuery$) and $i.skv < p_2.lb$ during the invocation of $p_2.processQuery$. In this situation,

p_2 would detect the discontinuity in the range satisfied by p_1 and its own range and would reject the query at line 2 of *processQuery*. Since the query terminates successfully, it must be the case that there exists a peer p such that $i \in p.range$ for the operation $p.processQuery$. Therefore $i \in R$. This proves Condition 2 of the Correct Query Result definition. □

Algorithm 2. $p.rangeQuery(lb, ub, initiator)$

```
1. if lb ∉ p.range then
2.    Reject query
3. else
4.    r := (lb, ub] ∩ p.range [1]
5.    items := items in p.items that are in range r
6.    Send items to initiator
7.    if ub ∉ p.range then
8.       Invoke succ(p).processQuery(p.ub, ub, initiator) asynchronously
9.    end if
10. end if
```

Algorithm 3. $p.processQuery(lb, ub, initiator)$

```
1. if lb ≠ p.lb then
2.    Reject query
3. else
4.    r := (lb, ub] ∩ p.range [1]
5.    items := items in p.items that are in range r
6.    Send items to initiator
7.    if ub ∉ p.range then
8.       Invoke succ(p).processQuery(p.ub, ub, initiator) asynchronously
9.    end if
10. end if
```

4.3 Incorrect Query Results Revisited

For both examples of incorrect range query results given in Section 3, if the simple protocol is used, some peer will reject the query. In the case of the inconsistent successor pointers, p_4 will forward the query with range $[5, 10]$ to p_1. p_1 will then reject the query because the lower bound of the range query, 5, does not equal $p_1.lb$, which is 6. For the case of concurrent range redistribution, p_2 will forward the query [15, 18] to p_3. p_3 will reject the query because the lower bound of the query, 15, does not equal its lower bound, 16.

[1] The items may be returned in parallel with the forwarding of the query to $succ(p)$.

5 Extension to Simple Protocol

One drawback of the scheme described above is that the system cannot satisfy a range query if that range encompasses a portion of the index that is concurrently in transit from one peer to another as a result of a redistribution. This issue exists even if there are no topology changes in the system. In this section, we describe an extension to the simple technique that overcomes this drawback. We show that in a system with no topology changes, the extended protocol can satisfy all queries correctly.

Suppose that peer p_2 relinquishes part of its range to its predecessor p_1 as a result of a *redistribute* operation. Instead of deleting the items that p_2 has given up to p_1, p_2 marks the items as *relinquished* and keeps them for some period of time. What can p_2 safely do with the relinquished part of its range? It cannot accept any inserts or deletes for this range because it is no longer the owner of the range and cannot guarantee that the future owner of the range, p_1, can consistently incorporate these insert and delete operations. p_2 also cannot accept *lookup*(*key*) operations for the relinquished range because it may report that no item exists for a given key even if an insert for this key has been successfully completed at p_1. Similarly, p_2 may return an item in response to a lookup after that item has been successfully deleted from p_1. p_2 can, however, use this relinquished range to satisfy range queries that have been forwarded from p_1. If p_1 forwards a query to p_2 with a lower bound equal to the lower bound of the relinquished range, then p_1 has not yet assumed responsibility for this range, and therefore no new items have been inserted into nor have any items been removed from the range. So, p_2 can return items in the relinquished range with no risk of omitting any live items and no risk of returning any deleted items.

The extended query protocol that uses this approach is given below. As in the simple protocol, the query begins with a *lookup*(*lb*) operation. *rangeQuery* in Algorithm 2 is invoked at the peer responsible for *lb*. The *processQuery* algorithm given in Algorithm 4 is invoked by each subsequent peer that participates in the query. The relinquished range is denoted by $p.range^* = (p.lb^*, p.ub^*]$. The set of items in $p.range^*$ is denoted $p.items^*$.

The question arises as to how long p_2 needs to keep the relinquished range and items. p_2 can delete the relinquished range and items once it knows that p_1 has received them. This confirmation can come in the form of an explicit acknowledgment message from p_1. Additionally, if p_2 receives a query forwarded by p_1 with the lower bound equal to $p_2.lb$ and not $p_2.lb*$, p_2 no longer needs to store the relinquished range. Finally, since p_2 is storing the relinquished range to answer queries on behalf of p_1, if p_2 detects that p_1 has failed or if p_2 is notified that it has a new predecessor, it no longer has any use for the relinquished range. Note that p_2 can delete the relinquished items at any time, and after the deletion, query processing becomes identical to the simple protocol.

We now prove the correctness of the extended protocol.

Algorithm 4. $p.processQuery(lb, ub, initiator)$

```
1. if lb ≠ p.lb and lb ≠ p.lb* then
2.    Reject query
3. else
4.    if lb = p.lb* then
5.       r* := (lb, ub] ∩ p.range*
6.       items := items in p.items* that lie in range r*
7.       lb := p.ub*
8.    end if
9.    if ub > p.lb then
10.      r := (lb, ub] ∩ p.range
11.      items := items∪ items in p.items that lie in range r
12.   end if
13.   Send items to initiator[2]
14.   if ub ≥ p.ub then
15.      Invoke succ(p).processQuery(p.ub, ub, initiator) asynchronously
16.   end if
17. end if
```

Theorem 2. *Using the extended protocol, every query that terminates successfully without being rejected produces a correct query result.*

Proof Sketch. The extended protocol is identical to the simple protocol except for the case where a range redistribution between a peer p and its predecessor $pred(p)$ takes place concurrent with the processing of a range query at $pred(p)$ and p.

Consider a query $Q = [lb, ub]$ that begins with operation o_s and ends successfully with operation o_e. As in the simple protocol, the operations *rangeQuery* and *processQuery* insure that, if the query terminates without rejection, the intervals $r \cup r^*$ satisfied at each of the peers that participate in the query are non-overlapping. The union of these intervals is exactly equal to $[lb, ub]$. Therefore, $\forall i \in R\ satisfies_Q(i)$ is true.

Let i be any item returned by peer p (i.e. $i \in R$ and $satisfies_Q(i)$ holds). i was either in $p.range$ or in $p.range^*$ during the execution of $p.processQuery$ (or $p.rangeQuery$). If $i \in p.range$, then Condition 1 for a Correct Query Result holds by the same argument given in the proof of Theorem 1. If $i \in p.range^*$ then $insertItem(i)$ happened before p sent i to $pred(p)$ in a *redistribute* operation. And $pred(p).processQuery$ happened before $pred(p)$ received items from the *redistribute* (otherwise, $pred(p)$ would have forwarded the query with lb equal to $p.lb$). So, $pred(p).processQuery \leq_{\mathcal{H}} redistribute \leq_{\mathcal{H}} p.processQuery$. If i was deleted, then $pred(p)$ must have received p's relinquished range before the $deleteItem(i)$ operation so $insertItem(i) \leq_{\mathcal{H}} redistribute \leq_{\mathcal{H}} deleteItem(i)$.

[2] As in the simple protocol, the items may be returned in parallel with the forwarding of the query to $succ(p)$.

Taking $o' = redistribute$, $live_{\mathcal{H}_{o'}}(i)$ is true, and therefore Condition 1 for a Correct Query Result holds.

The proof of Condition 2 is similar to that given in the proof of the simple protocol. □

This extended protocol greatly increases the system's ability to satisfy queries. In fact, in a system with no topology changes, the extended protocol can satisfy all queries correctly. We prove this by first showing that in a stable ring every $lookup(key)$ will eventually complete successfully . Therefore, for a range query $Q = [lb, ub]$, it is always possible to locate the peer responsible for lb. We then show that if a peer keeps the relinquished range until it receives confirmation that the range has been assumed by its predecessor, no query will ever be rejected.

Lemma 1. *In a stable ring with consistent successor pointers having only item insertions, item deletions, and range redistributions every lookup(key) will eventually terminate successfully.*

Proof. Suppose that due to outdated routing information, a $lookup(key)$ message is forwarded to a node p after p has transferred the range containing key to its predecessor. When p receives the lookup request, it detects that it is not the owner of key and forwards the request either to its successor or to some other peer in its routing structure. It can be shown that eventually some peer will be permanently responsible for the range containing key. At that time, the lookup must terminate successfully. In any redistribution, a node p with range $(p.lb, p.ub]$ transfers some portion of that range, $(lb, new\_lb]$, to its predecessor, and p becomes responsible for the range $(new\_lb, ub]$. Since the key space is discrete, eventually there is some node that is responsible for the range $(ub-\delta, ub]$ with $key \in (ub - \delta]$ such that no further subdivision of the range $(ub - \delta, ub]$ is possible. In this case, the only way for the peer to give up responsibility for this range is to leave the system. This is impossible under the assumption that the system topology is stable. □

Theorem 3. *In a stable ring with consistent successor pointers having only item insertions, item deletions, and range redistributions, every query can be answered correctly using the extended protocol.*

Proof. Theorem 2 shows that any query that is not rejected produces a correct query result. Here we show that in a stable ring, no query will ever be rejected. By Lemma 1, every $lookup(lb)$ eventually arrives at the peer responsible for the lb. The first step in a range query with range predicate $[lb, ub]$ is a $lookup(lb)$ operation and therefore in $p.rangeQuery$ the $Reject$ statement at line 2 will never be invoked. A query will be rejected at line 2 in $p.processQuery$ if lb is not equal to either the lower bound of $p.range$ or the lower bound of $p.range^*$. In the case that there is a concurrent redistribution for which $pred(p)$ has not yet received the new range and items, lb will be equal to $p.lb^*$. In all other cases, any query forwarded by $pred(p)$ will be equal to $p.lb$. So, if a query is rejected at line 2, then it must have been forwarded to p by some peer other than $pred(p)$.

This can only occur if that peer has an inconsistent successor pointer, which is impossible under the assumption of the theorem. □

6 Conclusion

In this paper, we have presented two lock-free techniques for range queries in P2P systems that provably guarantee correct query results, a simple protocol and a more robust extension. Both techniques have the benefit of simplicity in analysis and implementation. Additionally, in a system with no topology changes, the extended technique can satisfy every range query correctly. This work represents an initial step towards better formalization and understanding of P2P systems.

References

1. Linga, P., Crainiceanu, A., Gehrke, J., Shanmugasundaram, J.: Guaranteeing correctness and availability in p2p range indicies. In: SIGMOD. (2005) 323–334
2. Ratnasamy, S., Francis, P., Handley, M., Karp, R.M., Shenker, S.: A scalable content-addressable network. In: SIGCOMM. (2001) 161–172
3. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware. (2001) 329–350
4. Stoica, I., Morris, R., Karger, D.R., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM. (2001) 149–160
5. Zhao, B.Y., Huang, L., Stribling, J., Rhea, S.C., Joseph, A.D., Kubiatowicz, J.D.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **Vol. 22, No. 1** (2004) 41–53
6. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM. (2004) 353–366
7. Crainiceanu, A., Linga, P., Machanavajjhala, A., Gehrke, J., Shanmugasundaram, J.: P-ring: An index structure for peer-to-peer systems. Cornell University Technical Report (2004)
8. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: VLDB. (2004) 444–455
9. Gupta, A., Agrawal, D., El Abbadi, A.: Approximate range selection queries in peer-to-peer systems. In: CIDR. (2003) 141–151
10. Sahin, O.D., Gupta, A., Agrawal, D., El Abbadi, A.: A peer-to-peer framework for caching range queries. In: ICDE. (2004) 165–176
11. Karger, D., Lehman, E., Leighton, T., Levine, M., Lewin, D., Panigrahy, R.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: ACM Symposium on Theory of Computing. (1997) 654–663
12. Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with CFS. In: SOSP '01. (2001) 202–215