# Range Addressable Network: A P2P Cache Architecture for Data Ranges

A. Kothari D. Agrawal A. Gupta S. Suri
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
{kothari, agrawal, abhishek, suri}@cs.ucsb.edu

## Abstract

*Peer-to-peer computing paradigm is emerging as a scalable and robust model for sharing media objects. In this paper, we propose an architecture and describe the associated algorithms and data structures to support the execution of range selection queries over data scattered across a P2P network especially for resource discovery in grid environments.*

*We develop a distributed data structure referred to as a range addressable network that provides the following two quality-of-service guarantees: (i) the located peer is one with the smallest superset of the query range (important from the application perspective), and (ii) in a P2P network of $n$ peers, a query is routed through $O(\log n)$ peers before the intended peer is found (important from the system perspective).*

*Our preliminary experimental evaluation indicates that the range addressable network has desirable properties of scalability and load-balancing, which are crucial for the success of a large-scale P2P system.*

## 1. Introduction

Peer to peer (P2P) computing has attracted enormous interest recently, both from the commercial and the academic communities. The underlying principle of P2P systems is very simple. A user wishing to participate in a P2P system registers his/her machine and, once registered, becomes a peer node. If a user at a peer node wants to search for a file, the user submits a query string (name of a file), and the system returns to the user the name of the peer that contains the file (if it is available in the system). Napster [16] became an overnight sensation as millions of users found it useful to share their music files. Its centralized index was technically deficient and not designed to scale to the large population that it found itself serving. Soon thereafter, however, other more decentralized file-sharing systems like Gnutella [10] and Freenet [8] came along that eliminated the need for a centralized index. The popularity of P2P systems has also resulted in several research projects [3, 15, 5, 7, 18, 17, 19] addressing issues such as scalability, fault-tolerance, and security.

In their current form, P2P systems are still primarily used for sharing files (or media objects). Yet they possess the potential to become much more than file sharing systems. A grand vision of P2P computing is emerging in the context of *computational grids* [6]. The emerging grid architecture will combine all the information and other resources (data, storage, computing power) into a loosely connected but highly available, reliable, robust system. The *information service component* of the computational grid tracks the availability and attributes of a large number of resources that are geographically distributed and heterogeneous in nature. A fundamental functionality of the information service is to locate resources with specific combinations of attribute values. Andrzejak and Xu [1] have recently proposed a P2P based distributed indexing infrastructure for indexing range attributes such as processing or storage capacity. A typical grid client would query for available hosts with memory capacities in the range of 256 MB to 2GB. In this approach, the authors extend the CAN systems such that peers in the hyperspace are responsible for value ranges instead of point values in domain. Extensions of simple object lookup functionality of P2P systems to support more general database query processing over P2P data are also being explored [11, 13, 2, 14, 12]. However the focus of these works is primarily on issues related to schema mediation and complex database operations.

In this paper, we extend the peer-based storage architecture for Computational Grid Systems by providing an architectural layer in which answers to the range queries are cached at the peers. We envision, that in a large Grid infrastructure, clients will often ask highly similar queries and burdening the underlying storage system to answer such repetitive questions will impede scalability. For example, systems such as NWS [20] and Globus [9] monitor grid resources and provide an API to its users to query about the status of such resources. Unfortunately, since these systems are centralized implementations high query workload can impact the performance significantly. Although the grid resource monitoring system proposed by Andrzejak and Xu is a distributed implementation, range query processing may require processing the query at multiple peers (since each peer is responsible for a specific range).

We have developed a P2P architecture, referred to as *Range Addressable Network (RAN),* for resource discovery in Grid systems. The results of prior range queries issued by grid clients are cached in RAN for future reuse. Our solution provides the following two quality-of-service guarantees:

1. from the application point-of-view, given a selection query we locate the peer which contains the smallest superset of the query range; and

2. from the system point-of-view, the path length for routing a query request is guaranteed to be $O(\log n)$, where $n$ is the maximum number of peers participating in the system.

As P2P systems mature and become viable platforms for distributed databases, there will be need to develop extended database query functionality. Our RAN architecture can be seen as an important step in that direction, for the following reason. Most query optimizers perform the selection operations in a given SQL query at the leaves of the query tree. Thus, `SELECTION` is one of the primitive operations that must be available in order to support complex query processing capabilities over P2P systems.

Our range addressable network has three main algorithmic and data structure components: (1) a *topology*, which determines the *logical* connectivity among the peers, (2) a peer management scheme, which handles the joining and departure of peers, and (3) a range management scheme, which partitions the data among active peers and performs range queries. In addition, we suggest several optimization techniques that increase the robustness and improve the load balance across the system. Our preliminary experimental evaluations indicate that the range addressable network has desirable properties of scalability and load-balancing, which are crucial for the success of a large-scale P2P system. In that RAN is an essential component that can be used to increase scalability and fault-tolerance of grid monitoring services such as NWS and Globus.

The paper is organized as follows. In Section 2, we present a graph-based structure to represent range interval information efficiently. The structure is referred to as *range addressable DAG*. In Section 3, we develop a variety of approaches to map the range addressable DAG over peers in a P2P network. In addition, we also identify opportunities and techniques for improving the performance of the P2P system based on range addressable network. In Section 4, we conduct an experimental evaluation to evaluate the performance of range addressable networks. We conclude with a discussion of our results in Section 5.

## 2. Range Addressable Network Topology

The proposed system consists of peers forming an overlay network with *range addressable DAG* topology described below. The data is stored at peers in the form of sets of relation tuples obtained by range selection queries over an attribute executing in the system. Peers may issue selection queries for certain ranges of values that an attribute can take. The system would try to locate the result of the selection by locating a peer using the overlay topology that stores all tuples required to answer the query. In case, no peer has the desired answer, the query is directed to the source(s). The computed result is then installed at a peer that is responsible for the corresponding selection range. Note that, the overlay can also be used as an index, in which case peers need not store the tuples themselves, but only the information that leads to peers having the tuples falling in a particular selection range.

The underlying topology of our architecture determines the *neighbor* relation among the active peers. Each peer maintains some information about its topological neighbors. These topological neighbors are completely logical, and do not imply any physical proximity. In our scheme, each peer has only a constant number of neighboring peers.

We assume that the tuples that are stored in our peer system are labeled $1, 2, \ldots, N$. A *range* $r = [a, b]$ is a contiguous subset of $\{1, 2, \ldots, N\}$, where $0 \leq a \leq b \leq N$. A range $r' = [a', b']$ is called a *superset* of $r$ if $a' \leq a$ and $b' \geq b$. The *size* of a range $r = [a, b]$ is its length, namely, $|b - a|$. Given a query range $[a, b]$, peers in RAN cooperate to find the *shortest superset* of $[a, b]$, if there is one. In the following subsections, we first develop a logical solution to index ranges over a key attribute for a database relation. Later, we develop a physical mapping of this solution over a P2P system.

### 2.1. A Naive Scheme

We first describe a simple tree topology, whose shortcomings help motivate our new range addressable topology. Imagine a balanced binary tree $T$ on the set of leaves $\{1, 2, \ldots, n\}$. With each node $v$ in the tree, we associate an *interval* $i(v)$, which is the range spanned by all the descendants of $v$. Thus, the interval of the root node is $[1, n]$; the left and the right children of the root have intervals $[1, \frac{1}{2}n]$

and $[\frac{1}{2}n + 1, n]$, respectively. See Figure 1 for illustration. The data structure is similar in effect to interval trees [4].

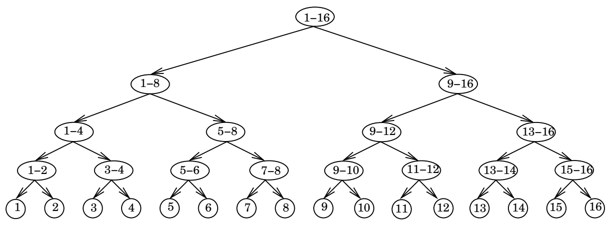

Figure 1: The basic tree topology on 16 leaves.

Given an arbitrary range $r = [a, b]$, let $v_r$ be the unique node of $T$ whose interval contains $r$ but the intervals of neither children of $v_r$ contain $r$. We call $v_r$ the *topology node* for $r$. It is easy to see that, for any range $[a, b]$, where $a, b \leq n$, there is a unique topology node for it. In the basic tree scheme, the range $r$ will be stored at node $v_r$. In Figure 1, for example, range $[3, 6]$ will be stored at the left child of the root.

The lookup for a query range $q = [x, y]$ can begin at any node. The search can move up or down in the tree, and so we initially set a boolean *down* true. Suppose we are at a node $v$. There are three cases to consider: (1) if the query range $q$ is not contained in $i(v)$, the search moves to the parent of $v$; (2) if $q$ is contained in the interval of a child of $v$, and *down* is true, then the search moves to that child; (3) Otherwise $q$ is contained in $i(v)$. If some range stored at $v$ is a superset of $q$, report it and stop. Otherwise, we set *down* to false, and the search moves to the parent of $v$.

The correctness of the search procedure follows from the simple observation that the interval of a node is divided among its two children, and the root's interval is the entire universe. It is also easy to see that the search will visit $O(\log n)$ nodes. However, this simple scheme suffers from a few significant drawbacks.

First, the search as outlined above *does not* always find the shortest superset of a query. As an example consider the query range $[7, 8]$. Suppose there are two ranges stored in the system that match it: $[7, 9]$ and $[2, 8]$. The topology node for $[7, 9]$ is the root, while $[2, 8]$ will be stored at the left child of the root. Assuming the lookup started at any leaf node between 1 and 8, the search will output $[2, 8]$ as the answer, because it will be found first. This example illustrates a key weakness of the basic tree topology—there is no correlation between the *size* of a range and its position in the tree. Arbitrarily small ranges can get mapped to nodes with arbitrarily large intervals. Specifically, the range $[\frac{1}{2}n, \frac{1}{2}n+1]$ is always stored at the root, as is any range that properly contains the leaf $\frac{1}{2}n$ in its interior.

While one can find the shortest superset in the basic tree topology by continuing the lookup all the way to the root, this can be undesirable for two reasons: (1) there is no way to know during the search if the shortest superset has already been found (adds inefficiency), and (2) the lookup forces all searches to go to the root (causes overload at the root). We solve both these shortcomings by using a directed acyclic graph (DAG) topology, which we describe next.

## 2.2. Range Addressable DAG

The range addressable DAG also maps the entire universe $[1, n]$ to a root node, but then recursively divides into three overlapping sub-intervals. Specifically, the root has three children nodes $v_1, v_2, v_3$, with intervals $[1, \frac{1}{2}n]$, $[\frac{1}{4}n + 1, \frac{3}{4}n]$, $[\frac{1}{2}n + 1, n]$, respectively. This recursive partitioning continues until each interval has length two, in which case we create two leaf nodes. See Figure 2 for an example. Observe that because of the overlapping intervals, a node can have up to 2 parents—thus, the topology structure is a DAG, not a tree. In terms of the number of levels, nodes and edges, however, the DAG has complexity similar to the basic tree. In particular, it can be shown that the range addressable DAG on $n$ leaves has at most $\log n + 1$ levels, and $O(n)$ nodes and edges (Lemma A.1). The mapping from

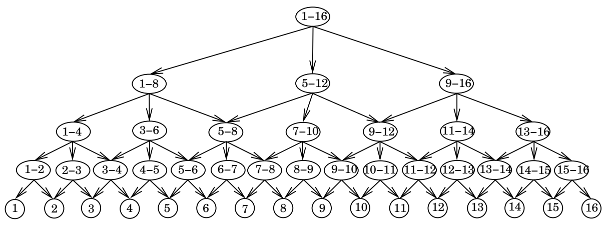

Figure 2: The range addressable topology DAG.

ranges to topology nodes is very similar to its counterpart in the basic tree. A range $r = [a, b]$ is associated with the unique DAG node $v_r$ whose interval $i(v_r)$ contains $r$, but none of the child-intervals of $v_r$ contain it. The lookup for a query range $q = [x, y]$ is slightly different, because the structure is a DAG, not a tree. Suppose the lookup begins at a node $v$. Initially, the boolean *down* is true. We again have the three cases to consider:

1. If $q \not\subseteq i(v)$, then the search moves to one of the parents of $v$ whose interval overlaps $q$;
2. If $q \subseteq i(w)$, for some child $w$ of $v$, and *down* is true, then the search moves to $w$.
3. If some range stored at $v$ is a superset of $q$, then we report the *shortest range containing $q$ that is stored at either $v$ or a parent of $v$,* and *stop.* Otherwise, we set *down* to false, and the search moves to one or both parents of $v$ whose intervals overlap $q$.

While the overall search scheme looks similar to the earlier scheme for basic topology, there are two key differences. First, because the search sometimes requires visiting *both* the parents of a node, the search complexity can potentially explode. It can be shown that this is not the case, and the lookup retains its $O(\log n)$ complexity (See Lemma A.4). Second, we will show that when a superset range is found, it is necessarily the *shortest superset* and hence the early termination in this search is correct. Thus, we avoid pushing the search up the hierarchy as soon as a match is found.

It may not be clear in Step 3 why the parent of a node can have a smaller range than the node itself. As an example consider a query range [5,8]. Suppose there are two ranges stored in the system that match it: [4,9] (stored at root in Figure 2) and [5,12] (stored at middle child of root). It can be shown that a parent is about as far as we need to search.

The two key properties of our range addressable DAG, namely, that a range of length $L$ is stored at a node whose interval length is close to $L$, and that a range query in this structure visits $O(\log n)$ nodes. In particular, if a range $[a, b]$ is stored at a node $v$ in the DAG, then the length of interval $i(v)$ is at least $|b-a|$ and at most $4|b-a|$ (See Lemma A.2). Thus, the range addressable DAG has the desired property that shorter ranges are stored near the fringe of the DAG, and only the extremely long ranges are stored towards the root. In addition, there is a well-defined relation between the *length* of a range, and its position in the DAG. We can establish the fact that our lookup finds the shortest superset of the query range, i.e., if $v$ is the lowest node in the DAG that contains a superset of the query range $q = [x, y]$ then, the shortest superset of $q$ is stored at either $v$ or a parent of $v$ (See Lemma A.3).

Because our lookup algorithm searches both $v$ (the lowest node with a range matching $q$) and its parents, we are guaranteed to find the shortest superset matching the query. Finally, we argue that our lookup scheme visits $O(\log n)$ nodes (See Lemma A.4). This guarantees that there are $O(\log n)$ nodes that need to be searched for the shortest superset range. Still, one needs to be careful in implementing the search described, because recursive calls to both parents can explode the search—the recursive calls can independently search the same set of nodes over and over. In our implementation, the lookup always goes to the left parent, who then sends a query to the right sibling if needed. Because the DAG has $O(\log n)$ levels, the search visits $O(\log n)$ nodes. We summarize these facts in the following theorem.

**Theorem 2.1** *The range addressable DAG with $n$ leaves has $O(n)$ nodes and edges, and $O(\log n)$ height. If a range is stored at a node of level $i$, then the range must have length at least $2^{i-2}$. Given a range selection query $q$, one can find the shortest superset of $q$ by searching $O(\log n)$ nodes in the worst-case.*

In the next section, we describe our peer protocol, which handles the mapping from the topology to peers and manages the peers in the system.

## 3. The Peer Protocol

In general, a P2P system managing data has to deal with the following problems. The system should be able to determine what parts of the logical structure are mapped to which peers. This mapping of peers to the logical data structure needs to be maintained dynamically as peers join and leave the system. The system should have a mechanism to locate the destination peers and route the queries to the destination peer. In addition, the system should deterministically be able to map the data ranges to specific peer(s). This is the key mechanism in speeding up query lookups.

In our scheme, the logical structure is a range addressable DAG with $N$ leaves, where $N$ is the number of values taken by the search attribute in the database. In the context of grid resource location, we can easily demonstrate that most attributes of interest can be categorically transformed to a finite value domain. For example, memory capacity can be mapped in terms of percentages, and similarly CPU availability can be mapped in terms of finite percentages.

Since the DAG has $N$ leaves, it follows from our discussion in section 2.2 that the lookup operation will be $O(\log N)$, which is undesirable. Latter in this section, we argue that the lookup operation can be done in $O(\log n)$, where $n$ is total number of peers in the system.

Our peer protocol has two important components: *peer management* and *range management*. The peer management component handles the joining or leaving of a peer. The range management component handles how the underlying database ranges are mapped to the current set of peers in the system. It also defines the routing protocol used by a peer to perform a lookup query.

### 3.1. Peer Management

The peer management component is responsible for handling the joining and leaving of peers. This component ensures that at any given time, the set of available peers partition the entire topology DAG among themselves; i.e., every node of the DAG is assigned to some peer. The set of nodes assigned to a peer is called its *zone*. The zone of a peer is always a *connected subgraph* of the original DAG and the union of all the zones is the entire DAG. The first peer to join the system has the entire DAG as its zone. As new peers join, the zones get redefined, but always form a partition of the DAG. The peer management component takes care of splitting and merging of these zones as peers dynamically join and leave the system. Two peers $p_1$ and $p_2$ are neighbors and keep information about each other if there is a parent-child relationship among any of the nodes in their respective zones.

In the range addressable DAG, a node can have two parents. We define a child node to belong to the zone of its *left parent*. Figure 3 shows an example partitioning of the DAG

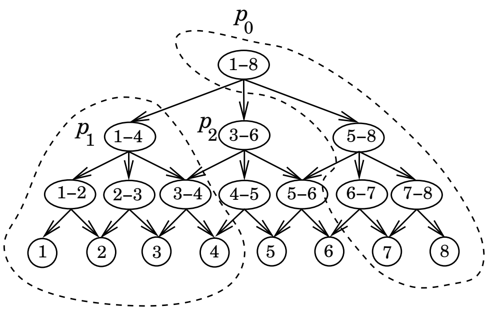

Figure 3: The zone of a peer. The zones of peers $p_0$ and $p_1$ are shown by dashed curves. The remaining part of the DAG forms the zone of $p_2$.

among 3 peers. We will use the terms *parent* and *child peer* to convey the relation between two neighboring peers.

In case of a P2P system, the query lookup measures the number of peer a query has to go through before finding a peer, which can service it. In a range addressable DAG, if a query is forwarded from one node to other, such that, both nodes belong to the same peer then the forwarding doesn't contribute to the query lookup. Therefore, the query lookup is no longer function of the size of the DAG but only depends upon how is the DAG divided among the peers.

Consider a *collapsed* DAG, where we collapse each peer's zone to a single node. It is easy to see that the lookup is $O(h)$, where $h$ is the height of the collapsed DAG. We call an $n$-peer system to be *balanced* if the range addressable DAG is divided among the peers in such a way that the corresponding collapsed DAG has a height of $O(\log n)$. A balanced system is desirable and the peer management component should strive to achieve it.

**3.1.1. Join Requests.**The new peer $p_{\text{new}}$ discovers an existing peer $p_{\text{old}}$ by contacting a bootstrap directory server, and sends a join request. On response to the join request, $p_{\text{old}}$ hands out the sub-DAG (under its ownership) rooted at one of its children to the new peer $p_{\text{new}}$. Since each node has at most three children, $p_{\text{old}}$ can become (parent) neighbor of at most 3 other peers. By default, the first peer to send a join request to $p_{\text{old}}$ inherits the DAG rooted at the left child; the next inherits the DAG rooted at the middle child; and the third one inherits the DAG of the right child.

**3.1.2. Leave Requests.**When a peer leaves, its zone is handed over to one of its neighboring peers (either a parent or a child). In order to balance the zone sizes, we merge the leaving peer's zone with the neighbor that has the smallest zone. Note that the newly merged zone is still a connected DAG, preserving our scheme's invariant. In addition to the zone merging, we also need to modify the neighbor relation among the remaining peers. This cost is proportional to the number of neighbors of the leaving peer which is at most a constant.

**3.1.3. Failure Events.**The basic peer protocol, described so far, is susceptible to failures as any peer in the system knows only about a constant number of other peers (its parent and children) in the system. Also failure of a single peer can disconnect the DAG into two disjoint components such that peers in one component might not be able to reach peers in the other component. These problems can be solved by letting a peer maintain information regarding some other peers in the system. Therefore, we modify our peer protocol such that a peer not only maintains information about its parent but also about all of its ancestors. This information can be further used to reduce the time it takes for a query lookup. Instead of forwarding a query to its parent, a peer can directly forward the query to its ancestor whose sub-DAG contains the topology node corresponding to the query.

The failure recovery mechanism works as follows. During a query lookup, if a peer finds that its parent has failed, it sends a *zone take-over* request to its first alive ancestor. The ancestor checks whether some other peer has already taken over the zone or a part of it. If not, the requesting peer is allowed to take over the zone. In case some other peer has already taken over the zone, the requesting peer's ancestor list is updated and the process is repeated, where the peer talks to its new ancestor to take over the remaining part of the zone. For example, in Figure 4, let peers $p_1, p_2$ and $p_3$ be responsible for node (1), (4) and (1–8) respectively. Peer $p_1$ finds that its ancestors who were responsible for nodes (1–2) and (1–4) have failed. It sends a zone take over request to $p_3$. Since, no other peer has taken over the zone, $p_3$ allows $p_1$ to take over them. Later on peer, $p_2$ also notices that peer responsible for nodes (1–4) and (3–4) has failed. It also sends a zone take over request to $p_3$ but since $p_1$ has already taken over node (1–4), we update the ancestor list of $p_2$ and $p_2$ sends a zone take over request to $p_1$ to take over node (3–4).

## 3.2. Range Management

The range management component is responsible for mapping ranges to the peers. Since the logical structure we consider is a range addressable DAG, the mapping of ranges to the logical structure is straightforward. In order to map a

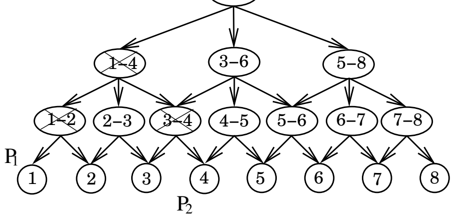

Figure 4: Failure Recovery. Peers responsible for crossed nodes have failed.

range to a peer, we consider the topology node corresponding to a range. The range is stored at the peer whose zone contains the topology node.

**3.2.1. Range Lookup.** Suppose the query begins at a peer node $p$. We use the algorithm described in Section 2.2 to move to a parent or a child peer of $p$, until we find a peer $p'$ that contains a range that is the superset of the query $q$. At this point, it is guaranteed from the property of RAN topology that the answer can be found at $p'$ or a parent of $p'$. It should be noted that during moving up or down the range addressable DAG, multiple levels of the DAG can fall in the same zone and hence will be controlled by the same peer. The query needs to be forwarded to a neighboring peer only when the traversal in the DAG crosses zone boundaries. In that case, the peers will be neighboring peers and have information about each other.

**3.2.2. Range Update.** As a result of database updates, tuples belonging to different ranges can get affected. In the absence of any control mechanism, the only alternative is to propagate the change to every peer in the system. Clearly, such an approach is not feasible for P2P systems where no single site has complete knowledge about the system. In the range addressable network, database and range updates can be handled easily as follows. When a tuple is updated, we search for the peer responsible for that tuple. The search locates the peer $p$ with the shortest range containing that tuple. Once again the property of RAN topology ensures that all other peers containing ranges with this tuple are among the ancestors of this peer $p$. We, therefore, propagate the update up the DAG through the left parent of $p$, who also notifies any right sibling that needs to be updated.

## 3.3. Improving System Performance

In this section, we discuss several techniques that can improve system performance via better load balancing and query routing. In particular, we discuss two techniques: *cross pointers*, which are additional links among the peers to provide shortcuts during the query routing; and *peer sampling*, which addresses load balancing by finding peers with large zones to split .

**3.3.1. Improved Routing and Robustness through Cross Pointers (CP).** We can improve query routing in the network by adding some well-placed cross pointers among the peers. In Figure 5, the link from node (3–4) to (5–6) is an example of a cross pointer. When cross pointers are present, queries can be routed faster, since the queries can be forwarded within a given level of the DAG without going through the hierarchical route.

In particular, if a node $v$ is the left child of its parent, then it keeps cross pointers to all the left children of nodes that

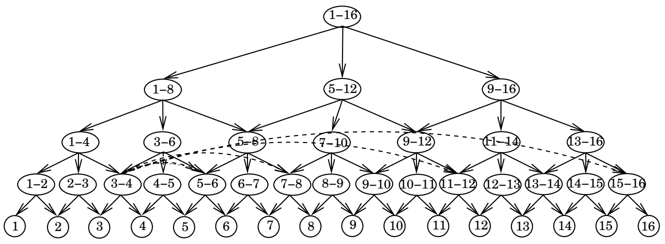

Figure 5: Range Addressable DAG with Cross Pointers

are in its parent's level. Similarly, if $v$ is the middle child of its parent, then it may keep cross pointers to all the middle children of nodes that are in its parent's level. Note that a cross pointer needs to be stored at a peer only if it points to a topology node in other peer's zone.

The cross pointers also improve the robustness of the system by providing alternate routes between any two peers in the system. A substantial number of these paths will be disjoint, which ensures that in case of failures, with high probability a path will exist between two peers .

**3.3.2. Load Balancing by Peer Sampling.** As discussed earlier in the section, a balanced system is desirable because of its optimal lookup performance. Therefore when a new peer joins, we want it to split the zone with another peer, $p$, such that the height of collapsed zone remains same. This is in general impossible to ensure in the worst-case, since without a centralized server, the identity of $p$ might be unknown to other peers. We suggest and implement the idea of *peer sampling* to poll a small number of peers to determine which peer's zone should be split with the newly joining peer. When a new peer $p_{\text{new}}$ arrives, it randomly polls $k$ peers in the system, where $k$ is a tunable parameter. Among these $k$ polled peers, $p_{\text{new}}$ chooses to join the one whose zone is rooted closest to the root.

## 4. Experimental Evaluation

We have performed simulations to evaluate the performance of our scheme and compare the relative merits of the different policies and techniques proposed in this paper. We have used different metrics in order to evaluate the performance from the perspective of system as well as applications. The primary performance metric from an application's perspective is the *latency* of answering a range query. We measure latency in terms of *route length*, which is the number of peers through which a query was routed in the P2P overlay network, before the intended peer was located. From a system's perspective, the quality of our scheme can be measured by the *query load* experienced by various peers in the system. The query load can be further divided into a *query forwarding* load, which measures the number of range queries a peer forwards to its neighbors, and a *query processing* load, which measures the number of range queries answered by the peer.

In Section 3.3, we proposed two techniques to improve the performance of the basic scheme: cross pointers and peer sampling. The cross pointers only effect the route length and query forwarding load. For both of these metrics, we compare the basic scheme (BS) with the modified scheme that uses cross pointers (BS-CP). In all our simulations we have used peer sampling, where each peer samples a constant number of peers before joining the system.

In all our experiments, we consider a database of size $2^{23}$ tuples. The set of range queries has been generated by picking query ranges uniformly at random from the set of all possible range queries within range lengths of $2^8$ and $2^{13}$. In addition, we assume that the data is distributed uniformly. Peers submit queries uniformly at random to the system.

We also evaluate the behavior of the system in presence of failures. We measure the robustness of our scheme using the *query success rate* and *failure messages*. The query success rate measures the fraction of queries, which are answered using the cached queries. The failure messages measures the number of extra messages a peer has to process in order to maintain the logical structure in presence of failure.

## 4.1. Route Length

In this experiment, we measure the average route length as the number of peers grow in the system. The number of peers are varied in powers of two: from $2^8$ to $2^{13}$. Figure 6 shows the result for the experiment, which is averaged over $15$ runs. In Section 3.1, we argued that in a balanced system the route length is $O(\log n)$ where $n$ is the number of peers in the system. In case of BS, the average path length increases logarithmically with the number of peers in the system. This implies that our peer joining strategy along with peer sampling achieves a balanced system. The experiment also clearly validates the usefulness of cross pointers, where route length is reduced significantly. In addition, with cross pointers the route length is not much affected by the growing number of peers in the system.

## 4.2. Query Load

The experiment measures the query load as the number of queries vary in a system with 10000 peers. We varied the number of queries from $10^5$ to $10^6$ in steps of $10^5$. The values shown in the graphs are averaged over $15$ runs. The cross pointers only affect how the query is routed to the destination peer but has no impact on the query processing load of a peer. Therefore, we only consider them in case of query forwarding.

Figure 7 plots the percentage of queries processed of the peer with the maximum load as the number of queries increases. It is interesting to notice that irrespective of total number of queries, the maximal loaded peer always process around $0.22\%$ of the total queries.

The next plot, Figure 8, shows the average forwarding load on peers. Using cross pointers reduces the load by a factor of 3. In Figure 9, we compare the forwarding distribution for BS and BS-CP for the case with $10^6$ queries. The distribution for BS-CP is more even with more than 90 percent of the peers having a forwarding load between 64 and 4096 queries. Thus, the cross pointers not only help in reducing the load but also in distributing them more uniformly.

## 4.3. Failure

The experiment evaluates the performance of the failure recovery mechanism under a worst case failure scenario. In the simulation, first we run $10000$ range queries, then we induce failure of $x$ percent of random peers and run another $90000$ range queries. We experimented with $5$ different values of $x$: $1,\ 5,\ 10,\ 20$ and $50$. The reported results are averaged over $12$ runs.

In the absence of failures, a large fraction of queries can be answered by the cached queries. Failures increase the number of cache miss because of two reasons: First, due to

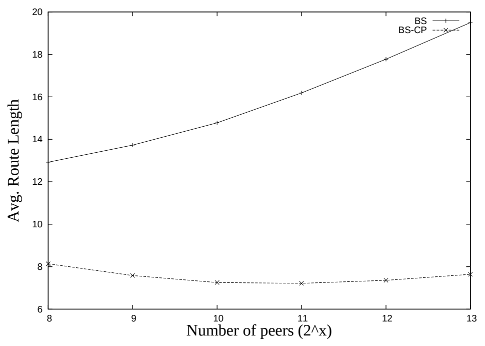

Figure 6: Average route length for the four variants of our scheme.

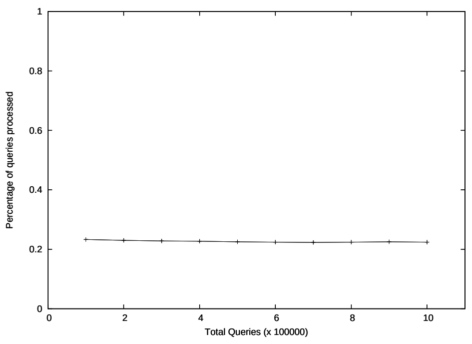

Figure 7: Query Processing Load: Percentage of queries processed by maximum loaded peer.

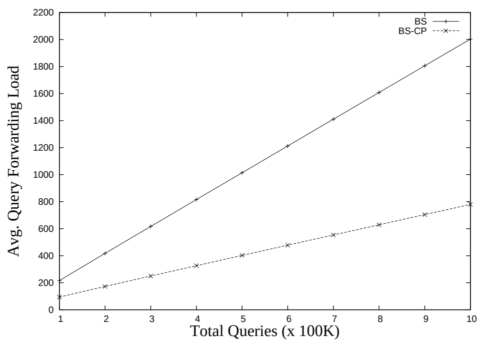

Figure 8: Average query forward load.

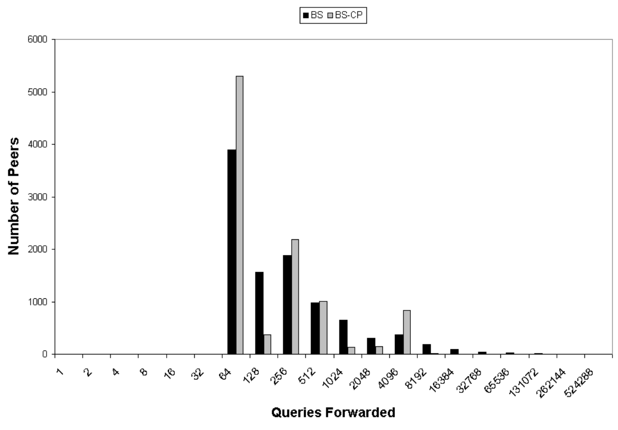

Figure 9: Distribution of query forwarding load for a set of 1M queries.

failures, some of cached ranges are lost and second, failures might temporarily disconnect the structure due to which a peer might be unable to access certain cached ranges. In our simulation scenario, the second case is aggravated because of simultaneous failure. In real world, the failures are more gradual and the failure handling mechanism has more time to recover.

Let $QA(x)$ be the number of queries answered when $x$ percent of peers have failed. We cannot judge a failure recovery mechanism solely based on $QA(x)$ because some range query sequence can have a lot of cache misses even in absence of failures. Therefore, we evaluate the performance using the ratio $QA(x)/QA(0)$, which we call as *query success rate* . Figure 10 plots the query success rate as the value of $x$ increases. The high query success rate even in case of high failure rates, indicates the usefulness of our failure recovery mechanism.

Figure 11 plots the maximum number of overhead messages processed by any peer. The number increases with the increase in failure rate but even for the higher failure rates, the value is around 30 messages, which is neglicompared to the query forwarding and processing load a peer would have to bear.

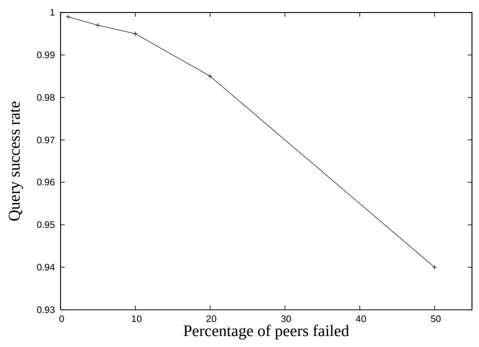

Figure 10: Query success rate in presence of failure.

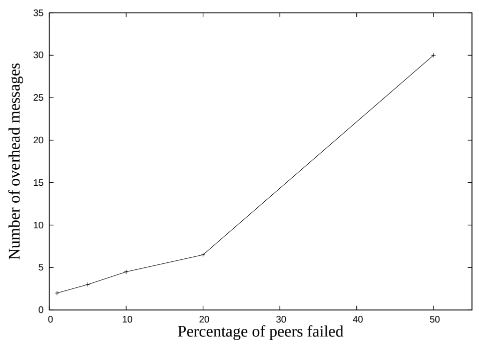

Figure 11: Failure messages processed by a peer due to failure. We plot the maximum number of message processed by any peer.

## 5. Discussion

P2P systems have become a prevalent technology to share media objects over a wide-area network. Several commercial P2P systems are already in use and research prototypes are underway to address the scalability, performance, and fault-tolerance issues associated with commercial P2P systems. However, the functionality of commercial systems and research prototypes is limited to providing object lookup in a distributed manner. In that, such systems basically support distributed directory service for file-based objects scattered over a wide-area network.

In this paper, we explore the possibility of using the P2P paradigm to design a large-scale data sharing architecture with limited database query processing capabilities which will be a useful middleware for grid computing applications. The ultimate goal is to design a P2P database architecture in which data is scattered over the peers, and peers can access such data by issuing SQL-like queries. As a first step towards building such an architecture, we present a design of a distributed data-structure, referred to as a *range addressable network*, that facilitates range query lookups and range query processing. This data structure is based on a logical abstraction of a directed acyclic graph and maintains enough information about ranges so that range lookups can be processed efficiently. The efficiency measure ensures that a range query is processed using a smallest superset of the query range (if one exists) and route length of the lookup request grows only logarithmically in the size of the network.

## References

[1] A. Andrzejak and Z. Xu. Scalable, efficient range queries for grid information services. In *Proceedings of the Second IEEE International Conference on Peer-to-Peer Computing (P2P2002)*, 2002.

[2] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In *Proceedings of the Fifth International Workshop on the Web and Databases (WebDB 2002)*, Madison, Wisconsin, USA, June 2002.

[3] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. In *Twelfth IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Press, June 2003.

[4] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*, volume 2. Springer, 1999.

[5] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. *Lecture Notes in Computer Science*, 2009:67–??, 2001.

[6] I. Foster and A. Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[7] M. J. Freedman and R. Morris. Tarzan: a peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 193–206. ACM Press, 2002.

[8] Freenet. http://freenet.sourceforge.net/.

[9] Globusi. http://www.globus.org/.

[10] Gnutella. http://gnutella.wego.com/.

[11] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can peer-to-peer do for databases, and vice versa? In *Proceedings of the Fourth International Workshop on the Web and Databases (WebDB 2001)*, Santa Barbara, California, USA, May 2001.

[12] A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research*, Asilomar, California, United States, January 2003.

[13] M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

[14] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proceedings of the 2003 ACM SIGMOD*. ACM Press, June 2003.

[15] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192. ACM Press, 2002.

[16] Napster. http://www.napster.com/.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[20] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems (to appear)*, 15(5), 1999.

## A. Appendix

**Lemma A.1** *The range addressable DAG on $n$ leaves has at most $\log n + 1$ levels, and $O(n)$ nodes and edges.*

**Proof:** The bounds on the number of levels and nodes follow from the fact that there are $2^{i+1} - 1$ nodes at distance $i$ from the root. The bound on the number of edges follows because there are at most $3(2^{i+1} - 1)$ edges between the nodes of levels $i$ and $i + 1$. ▮

**Lemma A.2 (Range to Node Mapping)** *If a range $[a, b]$ is stored at a node $v$ in the range addressable DAG, then the length of interval $i(v)$ is at least $|b-a|$ and at most $4|b-a|$.*

**Proof:** By definition, if $[a, b]$ is stored at $v$, then the length of $i(v)$ cannot be smaller than $|b - a|$. Thus, we only need to show that $i(v)$ is at most $4|b - a|$. Consider the partition of $i(v)$ into four equal parts, and call these sub-intervals $\sigma_1, \sigma_2, \sigma_3$ and $\sigma_4$. By construction, the intervals associated with the three children of $v$ are $\sigma_1 \cup \sigma_2$, $\sigma_2 \cup \sigma_3$ and $\sigma_3 \cup \sigma_4$. Because $[a, b]$ is stored at $v$, and not at its children, it must be the case that $[a, b]$ is not contained in the union of any two consecutive $\sigma_j$'s, for $j = 1, 2, 3, 4$. Thus, $|b - a|$ must be strictly longer than any $\sigma_j$. But that guarantees that $|b-a| > |\sigma_j| = \frac{1}{4}|i(v)|$, which proves the lemma. ▮

**Lemma A.3 (Shortest Superset)** *Suppose $v$ is the lowest node in the DAG that contains a superset of the query range $q = [x, y]$. Then, the shortest superset of $q$ is stored at either $v$ or a parent of $v$.*

**Proof:** Suppose $[a, b]$ is a range stored at $v$ that matches $q$. We show that no ancestor higher than $v$'s parent can contain a range that matches $q$ and has length smaller than $|b - a|$. Let $w$ be a grandparent or higher ancestor of $v$. Then, since the interval length doubles at each level of DAG, we have $|i(w)| \geq 2^2|i(v)|$. By Lemma A.2, the shortest range stored at $w$ has length strictly bigger than $|i(w)|/4$. On the other hand, since $[a, b]$ is stored at $v$, $|b - a| \leq |i(v)|$. Thus, if $r$ is any range stored at $w$, then it must be that $|r| > \frac{1}{4}|i(w)| \geq |b - a|$. ∎

**Lemma A.4 (Range Ancestors)** *Consider a node $v$ in the range addressable DAG and its interval $i(v)$. At any level of the DAG, there are at most two ancestors of $v$ whose intervals overlap with $i(v)$. In addition, these ancestors are mutual siblings, the left ancestor is reachable from the left parent of $v$.*

**Proof:** The key observation in establishing this lemma is this: at any level of DAG, the intervals of any two *non-sibling* nodes are *disjoint*. This follows from the DAG construction. Thus, any two nodes whose intervals overlap $i(v)$ must be siblings. If a node has two parents, then its interval overlaps with that of its left parent. Inductively, this gives a path to the left ancestor of $v$. ∎