Query Processing Over Peer-To-Peer Data Sharing Systems

O. D. Şahin A. Gupta D. Agrawal A. El Abbadi
Department of Computer Science
University of California at Santa Barbara
{odsahin, abhishek, agrawal, amr}@cs.ucsb.edu

Abstract Peer-to-peer systems are mainly used for object sharing currently, but they can provide the infrastructure for many other applications. In this paper, we extend the idea of object sharing to data sharing on a peer-to-peer system. We propose a method, which is based on the CAN [9] system, for efficiently evaluating range queries on such a system. The answers of the range queries are cached at the peers and then they are used to answer further range queries. The scalability and efficiency of our design is shown through simulation.

Keywords: Peer-to-peer Systems, Distributed Systems, Distributed Databases, Range Queries, Distributed Hashing.

1 Introduction

Peer-to-peer systems have been increasing in popularity in recent years as they are used by millions of users as they allow the exchange of massive amounts of data over the Internet. These systems are generally used for file sharing, such as Napster [8] and Gnutella [1], which allow users to share their files with other users. There are two challenges to be resolved for sharing objects on a peer-to-peer system:

- **Data Location:** Given the name of an object, locate the corresponding object.
- **Routing:** Once the possible location of the object is found, how to route the query to that location.

Napster uses a centralized design to resolve these issues. A central server maintains the index for all objects in the system. New peers joining the system

register themselves with the server. Every peer in the system knows the identity of the central server while the server keeps information about all the nodes and objects in the system. Whenever a peer wants to lookup an object, it sends the request (name of the object) to the central server which returns the IP addresses of the peers storing this object. The requesting peer then uses IP routing to pass the request to one of the returned peers and downloads the object directly from that peer. There are several shortcomings of the centralized design of Napster. First of all, it is not scalable since the central server needs to store information about all the peers and objects in the system. Second, it is not fault tolerant because the central server is the single point of failure.

A different approach is followed by Gnutella to get around the problem of centralized design. There is no centralized server in the system. Each peer in the Gnutella network knows only about its neighbors. A flooding model is used for both locating an object and routing the request through the peer network. Peers flood their requests to their neighbors and these requests are flooded for a certain threshold. The problems associated with this design are the high overhead on the network as a result of flooding and the possibility of missing some requests even if the requested objects are in the system.

These designs, including Napster, Gnutella, and some other variants are referred to as *unstructured* peer-to-peer systems [4, 7], because the data placement and network construction are done randomly in these systems. Another group of peer-to-peer designs are referred to as *structured* peer-to-peer systems and include systems such as CAN [9], and Chord [10]. These systems are based on implementing a distributed data structure called *Distributed Hash Table* (DHT) [9, 10, 11] which supports a hash-table like interface for storing and retrieving objects.

CAN [9] uses a d-dimensional virtual address space for data location and routing. Each peer in the system owns a zone of the virtual space and stores the objects that are mapped into its zone. Each peer stores routing information about *O(d)* other peers, which is independent of the number of peers, N, in the system. Each object is mapped to a point in d-dimensional space and then the request is routed toward the mapped point in the virtual space. Each peer on the path passes the request to one of its neighbors which is closer to the destination in the virtual space. The average routing path has $O(dN^{1/d})$ hops which is the lookup time for exact match queries. Chord [10] assigns unique identifiers to both objects and peers in the system. Given the key of an object, it uses these identifiers to determine the peer responsible for storing that object. Each peer keeps routing information about $O(logN)$ other peers, and resolves all lookups via $O(logN)$ messages, where N is the number of peers in the system.

Since peer-to-peer systems have emerged as a powerful paradigm for data sharing over the Internet, a natural question arises if the power of peer-to-peer systems can be harnessed to support database functionality over peer-to-peer sys-

tems. Indeed, several research initiatives are underway to answer this question. For example, Gribble et al. [2] in their position paper titled "What can peer-to-peer do for databases, and vice versa?" outline some of the complexities that need to be addressed before peer-to-peer systems can be exploited for database query processing. Similarly, in a recent paper Harren et al. [5] explore the issue of supporting complex queries in DHT-based peer-to-peer systems. Harren et al. report the implementation of database operations over CAN by performing a hash join of two relations using DHT. The underlying technique basically exploits the exact-name lookup functionality of peer-to-peer systems.

The work reported in this paper has similar goals as that of Harren et al., in that we are interested in supporting database query processing over peer-to-peer systems. Most data-sharing approaches designed for peer-to-peer systems are concerned with exact lookup of data associated with a particular keyword. Our contention is that in order to achieve the larger goal of data-sharing in the context of a DBMS over peer-to-peer systems, we need to extend the current peer-to-peer designs that only support exact name lookups to range searches. Range searches or range selection is one of the fundamental functionality that is needed to support general purpose database query processing. The main motivation for this is that the `selection` operation is typically involved at the leaves of a database query plan and hence is a fundamental operation to retrieve data from the database. Assuming that such data partitions of a relation are extensively replicated at the peers due to prior queries, we would like to retrieve the data from the peer-to-peer system instead of fetching it from the base relation at the data source. In this paper we propose to extend peer-to-peer systems to support more general queries on potentially more complex and more structured datasets. Unlike previous approaches for the design of distributed databases, our approach aims to support the management of loosely synchronized datasets that support more general queries in a peer-to-peer environment. Unfortunately, DHTs were designed for exact match queries. In this paper, we will extend the idea of object sharing to data sharing and propose a method for efficiently answering range queries on a peer-to-peer data sharing system. Our general long term goal is to support the various types of complex queries used by DBMSs so that general peer-to-peer data support can be a reality.

The rest of the paper is organized as follows: Section 2 formulates the problem. Section 3 introduces the basic concepts of our design, which is explained in detail in Section 4. The experimental results are presented in section 5. The last section concludes the paper and discusses future work.

2 Problem Formulation

Current peer-to-peer systems focus on object sharing and use object names for lookup. Our goal, on the other hand, is to design a general purpose peer-to-peer data sharing system. We consider a database with multiple relations whose schema is globally known to all peers in the system. The peers cooperate with each other to facilitate the retrieval and storage of datasets. A straightforward extension and application of object naming is to use the relation name to locate the data in the system. However, such an approach will result in large amounts of data being stored redundantly and often unnecessarily throughout the network. A more desirable approach is to use peers to store the answers of the previous queries. Whenever a new query is issued, the peers are searched to determine if the query can be answered from the prior cached answers. This is similar to the known database problem often referred to as *Answering Queries using Views* [6]. Since the problem of answering queries using views is computationally hard even in centralized systems, we will instead focus at a restricted version by extending the exact lookup functionality of peer-to-peer systems to the range lookup of a given dataset. Hence, our goal is to develop techniques that will enable efficient evaluation of range queries over range partitions that are distributed (and perhaps replicated) over the peers in a peer-to-peer system.

We assume that initially the database is located at a known site, or a known set of sites. All queries can be directed to this database. However, such a centralized approach is prone to overloading. Furthermore, the location of the data may be quite remote in the peer-to-peer network, and hence response time may be slow. Our goal is for the peers to cooperatively store range partitions of the database, which are later used to respond to user queries. Of course the challenge is how to track down where the various data range partitions are located. A straightforward approach would be to maintain a centralized index structure such as an interval tree that has the global knowledge about the locations of range partitions distributed over the network. However, such an approach would violate the key requirement of peer-to-peer systems, which is to ensure that the implementation is scalable, decentralized, and fault-tolerant.

Typically when an SQL query is formulated, a query plan is designed in the form of a query tree. A common optimization technique is to push the selection operations down to the leaves of the tree to minimize the data that has to be retrieved from the DBMS. A similar approach is used here to minimize the amount of data retrieved from other peers for range queries. Rather than retrieving all possible tuples from the actual database for each range query, the answers stored at the peers are searched to find a smaller set of tuples that is a superset of the query.

For example, if the answer of a range query $\langle 20, 35 \rangle$ for a given attribute is

stored at a peer, then future queries such as $\langle 25, 30\rangle$ can be answered using the result of $\langle 20, 35\rangle$. Since the range $\langle 20, 35\rangle$ subsumes the range $\langle 25, 30\rangle$, it is enough to examine the tuples in the result of $\langle 20, 35\rangle$, without any data retrieval from the database. In this way, less tuples are checked to compute the answer and all the tuples to be examined are retrieved directly from a single peer. This also decreases the load on the database since it is not accessed for every query.

The problem can now be stated as follows:

Problem. Given a relation R, and a range attribute A, we assume that the results of prior range-selection queries of the form $R.A(LOW, HIGH)$ are stored at the peers. When a query is issued at a peer which requires the retrieval of tuples from R in the range $R.A(low, high)$, we want to locate any peers in the system which already store tuples that can be accessed to compute the answer.

In order to adhere to the peer-to-peer design methodology, the proposed solution for range lookup should also be based on distributed hashing. A nice property of DHT-based approach is that the only knowledge that peers need is the function that is used for *hashing*. Once this function is known to a peer, given a lookup request the peer needs to compute the hash value locally and uses it to route the request to a peer that is likely to contain the answer. Given this design goal, a naive approach would be to use a linear hash function over the range query schema, i.e., a linear hash function over low, $high$, or both low and $high$. A simple analysis reveals that such a hash function will only enable exact matches of given range requests. However we are also interested in the results of the range queries that may contain the given range, i.e., the ranges that are a superset for the given query range lookup. In the following sections we develop a DHT approach that enables range lookups that are not exact matches. In [3], we use locality preserving hash functions for range lookups that are based on similarity and hence provide approximate answers to range queries. In this paper, however, our technique ensures that a range lookup will always yield a range partition that is a superset of the query range, if one exists.

3 System Model

Our system is based on CAN [9] and uses a 2d virtual space in a similar manner. Given the domain $[a, b]$ of a one dimensional attribute, the corresponding virtual hash space is a two dimensional square bounded by the coordinates (a, a), (b, a), (b, b), and (a, b) in the Cartesian coordinate space. Figure 1 shows the corresponding virtual hash space for a range attribute whose domain is [20,80]. The corners of the virtual space are $(20, 20)$, $(80, 20)$, $(80, 80)$, and $(20, 80)$.

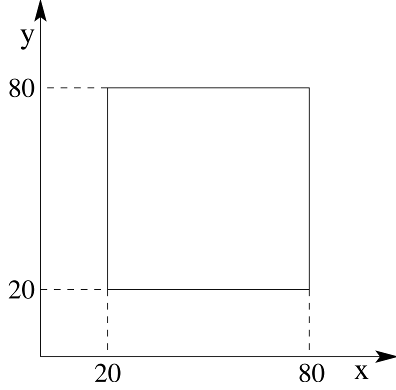

Figure 1: Virtual Range Lookup Space for a range attribute with domain $[20, 80]$

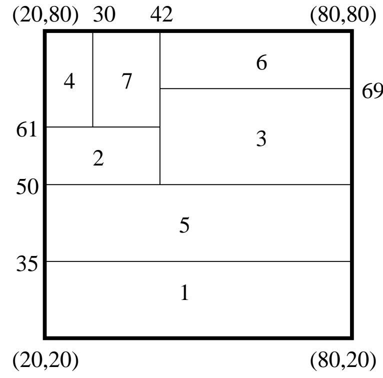

Figure 2: Partitioning of the virtual space shown in Figure 1

The virtual hash space is further partitioned into rectangular areas, each of which is called a *zone*. The whole virtual space is entirely covered by these zones and no two zones overlap. A zone can be identified by a pair $(\langle x_1, y_1\rangle, \langle x_2, y_2\rangle)$ where $\langle x_1, y_1\rangle$ is the bottom left corner coordinate whereas $\langle x_2, y_2\rangle$ is the top right corner coordinate. Figure 2 shows a possible partitioning of the virtual space shown in Figure 1. The virtual space is partitioned into 7 zones : *zone-1* $(\langle 20, 20\rangle, \langle 80, 35\rangle)$, *zone-2* $(\langle 20, 50\rangle, \langle 42, 61\rangle)$, *zone-3* $(\langle 42, 50\rangle, \langle 80, 69\rangle)$, *zone-4* $(\langle 20, 61\rangle, \langle 30, 80\rangle)$, *zone-5* $(\langle 20, 35\rangle, \langle 80, 50\rangle)$, *zone-6* $(\langle 42, 69\rangle, \langle 80, 80\rangle)$, and *zone-7* $(\langle 30, 61\rangle, \langle 42, 80\rangle)$.

Each zone is assigned to a peer in the system. Unlike original CAN, not all the peer nodes in the system participate in the partitioning. Those that participate are called the *active* nodes. Each *active* node *owns* a zone and stores the results of the range queries whose range hashes into the zone owned by this node. The rest of the peer nodes, which do not participate in the partitioning, are called *passive* nodes.

For the purpose of routing requests in the system, each *active* node keeps a *routing table* with the IP addresses and zone coordinates of its neighbors, which are the owners of adjacent zones in the virtual hash space. In Figure 2, the routing table of the owner of *zone-2* contains information about its four neighbors: *zone-5, zone-3, zone-7* and *zone-4*.

Given a range query with range $\langle q_s, q_e\rangle$, it is hashed to point (q_s, q_e) in the virtual hash space. This point is referred to as the *target point*. The target point is used to determine where to store the answer of a range query as well as where to initiate range lookups when searching for the result of a range query. The zone in which the target point lies and the node that owns this zone are called the *target zone* and the *target node*, respectively. The answer of each range query is stored

at the target node of this range. For example, according to Figure 2, the range query $\langle 50, 60 \rangle$ is hashed into *zone-3*, so the set of tuples that form the answer to this query would be stored at the node that owns *zone-3*.

Since the start point and end point of a range is hashed to x and y coordinates respectively, the y coordinate of the target point is always greater than or equal to the x coordinate. Hence , the target point never lies below $y = x$ line. Given two ranges $r_1 : \langle a_1, b_1 \rangle$, and $r_2 : \langle a_2, b_2 \rangle$ that are hashed to target points t_1 and t_2 in the virtual hash space, the following observations can be made:

1. If $a_1 < a_2$, then the x coordinate of t_1 is smaller than the x coordinate of t_2 and hence t_1 lies to the left of t_2 in the virtual space.

2. If $b_1 < b_2$, then the y coordinate of t_1 is smaller than the y coordinate of t_2 and hence t_1 lies below t_2 in the virtual space.

3. t_1 lies to the upper-left of t_2 if and only if range r_1 contains range r_2.

The third result can be concluded from the fact that by moving along the negative x direction in the virtual hash space decreases the start point of the corresponding range while by moving along the positive y direction increases the end point.

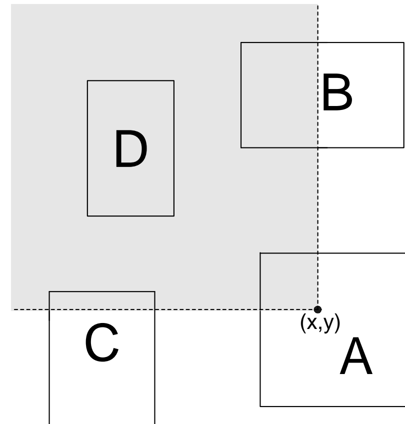

Figure 3: Range Hashing

Figure 3 shows a range query $\langle x, y \rangle$ that is hashed into zone A. Using the above observations, we can assert that if there is any prior range query result that contains $\langle x, y \rangle$, then it must have been hashed to a point in the shaded region. Any zone that intersects the shaded region is therefore a candidate for potentially containing a result for this query. In the figure, the zones A, B, C, and D intersect with the shaded region and may have a result that contains the given range

$\langle x, y \rangle$. Zone D is of particular interest since it is guaranteed to store results that completely contain the answer to the desired range $\langle x, y \rangle$.

Diagonal Zone. Consider a zone z bounded by coordinates $(\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle)$. We say that another zone z' bounded by $(\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle)$ is a diagonal zone of z if $a_2 \leq x_1$ and $b_1 \geq y_2$.

Intuitively, z' is diagonally above the upper-left corner of z. For example, D is a diagonal zone of A in Figure 3. If we require that zones cannot exist unless they are non-empty (i.e., store at least one range selection), then a diagonal zone of a zone z can answer all range queries that are hashed into z.

We now estimate the average routing distance for processing range queries in the proposed model. Let us assume that the 2d virtual hash space is partitioned into n equal sized zones. If $n = m^2$, then each axis is divided into m equal sized parts. Every path from a zone to another zone constitutes of steps in x and y directions (see Figure 4, the path is shown as a dotted line).

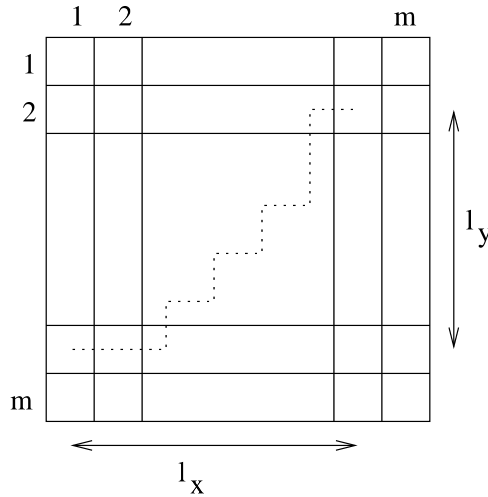

Figure 4: A Path in 2d virtual hash space divided into equal sized zones

In Figure 4, l_x is the number of steps in x direction and l_y is the number of steps in y direction. Then the total path length $l = l_x + l_y$. The shortest possible path length is 0 and the longest path length is $2m - 2$, which is $m - 1$ steps in either direction. Now, we will compute how many ways a path of length l can be generated. When $l < m$, $0 \leq l_x \leq l$ and $l_y = l - l_x$. A path of length l_x in the x direction can be generated in $m - l_x$ ways, and a path of length l_y in the y direction can be generated in $m - l + l_x$ ways. Hence the total number of ways in which a path of length l can be generated when $l < m$ is given by $S1$:

$$
\begin{aligned}
S1 &= \sum_{l_x=0}^{l} (m - l_x)(m - l + l_x) \\
&= -\frac{l}{6} + \frac{l^3}{6} - lm - l^2 m + m^2 + lm^2 \qquad (1)
\end{aligned}
$$

Therefore, the total number of paths whose length is less than m is:

$$
\begin{aligned}
E1 &= \sum_{l=0}^{m-1} S1 \\
&= \frac{m}{12} + \frac{7m^2}{24} + \frac{5m^3}{12} + \frac{5m^4}{24} \qquad (2)
\end{aligned}
$$

And, the weighted sum of all path lengths less than m is:

$$
\begin{aligned}
E2 &= \sum_{l=0}^{m-1} S1 \cdot l \\
&= -\frac{m}{30} - \frac{m^2}{12} - \frac{m^3}{12} + \frac{m^4}{12} + \frac{7m^5}{60} \qquad (3)
\end{aligned}
$$

When $m \leq l \leq 2m-2$, we have $l_x \leq m-1$ because that is the maximum distance that can be traveled in any direction. Since $l_y \leq m-1$, therefore, $l_x \geq l-m+1$. Hence the total number of ways in which a path of length l can be generated when $m \leq l \leq 2m-2$ is given by $S2$:

$$
\begin{aligned}
S2 &= \sum_{l_x=l-m+1}^{m-1} (m - l_x)(m - l + l_x) \\
&= \frac{l}{6} - \frac{l^3}{6} - \frac{m}{3} + l^2 m - 2lm^2 + \frac{4m^3}{3} \qquad (4)
\end{aligned}
$$

The corresponding total number of paths and weighted sum are given by the following expressions:

$$
\begin{aligned}
E3 &= \sum_{l=m}^{2m-2} S2 \\
&= -\frac{m}{12} - \frac{m^2}{24} + \frac{m^3}{12} + \frac{m^4}{24} \qquad (5) \\
and,\ E4 &= \sum_{l=m}^{2m-2} S2 \cdot l \\
&= \frac{m}{30} - \frac{m^2}{12} - \frac{m^3}{12} + \frac{m^4}{12} + \frac{m^5}{20} \qquad (6)
\end{aligned}
$$

The average path length $\bar{l}$ is therefore given by the weighted sum over all possible path lengths divided by total number of paths:

$$\begin{aligned} \bar{l} &= \frac{\sum_{l=0}^{m-1} S1.l + \sum_{l=m}^{2m-2} S2.l}{\sum_{l=0}^{m-1} S1 + \sum_{l=m}^{2m-2} S2} \\ &= \frac{E2 + E4}{E1 + E3} \\ &= \frac{2}{3}(m-1) \end{aligned} \quad (7)$$

The average path length in an equally partitioned hash space is therefore $O(\sqrt{n})$.

4 Distributed Range Hashing

In this section we describe the basic components that support the distributed implementation of range hashing. We assume that there are a set of computing nodes which participate in the distributed implementation of the range hash table(RHT). For simplicity, we are assuming that the range hash table is based on a relation R for a specific range attribute A with range extent $\langle low, high \rangle$. If queries on other attributes or relations also need to be supported, we assume a separate instance of an appropriate RHT will be maintained. The nodes participating in the system are in one of the two modes – *active/passive*. Initially, only one active node manages the entire virtual hash space. Other nodes become active as the work load on the active node increases. Next we describe how zones in the virtual hash space are maintained on peers. Finally, we present the details of range query lookup processing in the system.

4.1 Zone Maintenance

The partitioning of the virtual hash space into zones is at the core of both the data location and routing algorithms. Initially the entire hash space is a single zone and is assigned to one active node. The partitioning of the hash space is dynamic and changes over time as the existing zones split and new zones are assigned to passive nodes that become active and take responsibility for the new zone. A zone splits when it has a high load: it may have too many results to store (storage load) or it may get too many requests (processing load). The decision to split is made by the owner of the zone. Whenever a zone is to be split, the owner node contacts one of the passive nodes and assigns it a portion of its zone by transferring the corresponding results and neighbor lists. The split line, along which the zone

Algorithm 1 Split a zone

find x-median and y-median of the stored results
determine if a split at x-median (parallel to y axis) or a split at y-median (parallel to x axis) results in better partitioning of the space and then split along this line
compute new coordinates of this zone and the new zone according to the split line
assign the new zone to a passive node
for all results stored at this zone **do**
  if the result is mapped to the new zone **then**
    remove from this node and send to the new node
  end if
end for
for all neighbors of this zone **do**
  if it is a neighbor of the new zone **then**
    add it to the neighbor list of new node
  end if
  if it is no longer a neighbor of this node **then**
    remove from the neighbor list of this node
  end if
end for
add new node to the neighbor list of this node
add this node to the neighbor list of new node

splits into two, is selected by the owner node in such a way that results in an even distribution of the stored answers as well as an even distribution of the spatial space of the zone. The outline of the split operation is shown in Algorithm 1.

Figure 5 shows the partitioned zones after *zone-2* in Figure 2 splits parallel to the *y-axis*. Figure 6 shows the resulting partition if *zone-3* in Figure 5 splits parallel to the *x-axis*. The splitting operation does not guarantee that the two resulting zones will be equal in size or number of results, but it guarantees that each of the two zones will have at least one result after the split. This is unlike the original CAN design and we will show that it actually improves the performance. To satisfy the assertion that each zone has at least one stored result, some pre-computed results are stored initially in the system for the first zone.

4.2 Query Routing

When searching for the answer of a range query, the first place to look for cached results is the target zone of this range. Therefore whenever a range query is issued,

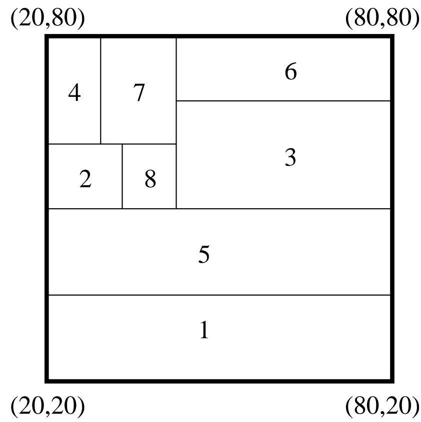

Figure 5: Partitioning of the virtual hash space after *zone-2* of Figure 2 splits

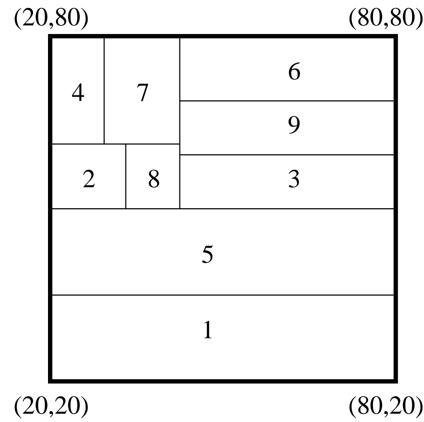

Figure 6: Partitioning of the virtual hash space after *zone-3* of Figure 5 splits

it is routed toward its target zone through the virtual space. Starting from the requesting zone, each zone passes the query to an adjacent zone until it reaches its target zone. Using its neighbor lists and the target point of the query, each node on the route passes the query to one of its neighbors whose coordinates are the closest to the target point in the virtual space. Algorithm 2 presents an outline of the routing algorithm.

Algorithm 2 Routing

```
if the query range maps to this zone then
  return this zone
else
  for all neighbors of this zone do
    Compute the closest Euclidean distance from the target point of the query
    to the zone of this neighbor in the virtual space
    if this is the minimum distance so far then
      keep a reference to this neighbor
    end if
  end for
  Send the query to the neighbor with minimum distance from the target point
  in virtual space
end if
```

A key step in Algorithm 2 is to choose which neighbor to forward the query. A simple way to compute the distance of the target from a zone is to compute the Euclidean distance between the target point and the center of the zone in the virtual hash space. In this case, a zone would compute the Euclidean distance

between the centers of the neighboring zones and the target point and forward the query to the neighbor with the least distance to the target. But as we show in the following discussion, the query might not always converge to the target.

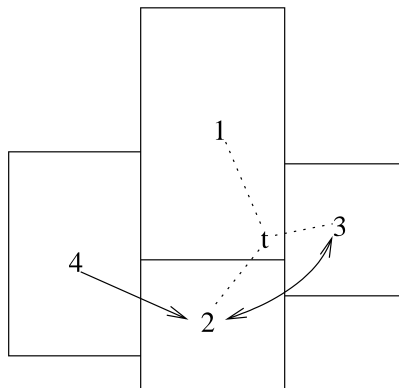

Figure 7: With distance measured from center of zones, queries may not converge

Figure 7 shows a portion of the virtual hash space that has been divided among the four zones labeled $1, 2, 3$ and 4. The labels of the zones denote the centers of the corresponding zones. The point t marks the target point in the virtual hash space and t lies in *zone-1*. The query originates in *zone-4* and is forwarded to *zone-2* because the center of *zone-2* is closer to the target t than that of *zone-1*. Once the query is at *zone-2*, it chooses *zone-3* among neighboring zones $1, 3$ and 4 because t is closest to center of *zone-3*. Next, *zone-3* sends it back to *zone-2* because between zones 1 and 2, the target t is closer to the center of *zone-2*. And the query keeps oscillating between zones 2 and 3 never reaching *zone-1* to which t belongs.

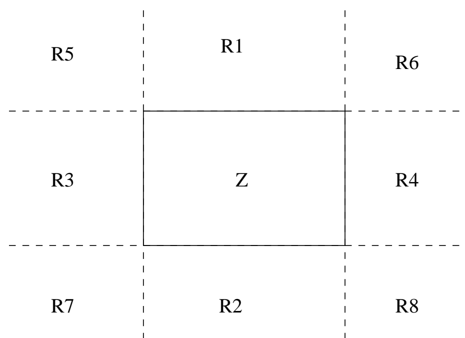

Figure 8: Measuring distance of a target point form a zone

The distance of target t from a zone Z should be measured as the closest distance of t from the entire zone. This is illustrated in Figure 8. The target point t may lie in one of the eight regions labeled $R1, R2, \ldots, R8$. If t lies in region $R1$ or $R2$ then the closest distance of t from the zone is its distance from the *upper* or *lower* edge of the zone rectangle respectively. Similarly, if t lies in region $R3$ or $R4$ then the closest distance is from the *left* or *right* edge of the zone rectangle respectively. If the target t is in one of the regions $R5, R6, R7$ or $R8$, then the closest distance of t from the zone is its distance from the closest vertex of the zone rectangle. If t lies in the interior of the zone or on the zone boundary, then its distance from the zone is 0. Since the target t can lie in the interior of at most one zone that zone will have the least distance from t and all other zones will have a positive distance from t. Since, at each step of routing the query moves closer to the target, it will converge. In case the target point is on an edge of a zone, then the query corresponding to that point should be stored in both the zones sharing that edge.

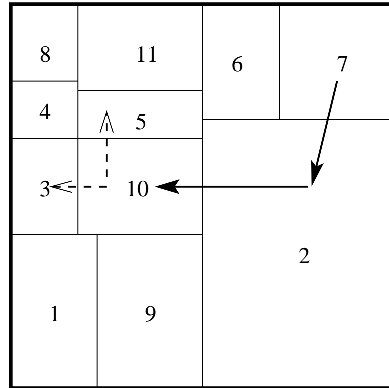

Figure 9: Routing and forwarding in the virtual hash space

Figure 9 shows how a query is routed in the system. The range query is initiated at *zone-7* and then routed through *zone-2* to its target zone, *zone-10*. The range queries in the system can be initiated from any zone. Since passive nodes do not participate in the partitioning, they send their queries to any of the active nodes from where the queries are routed toward the target zone.

4.3 Forwarding

Once a query reaches the target zone, the stored results at this zone are checked to see if there are any results whose range contains the query range. If such a result is found, only this result is examined to construct the query answer. Even if there is

no such local result, it is still possible that some other zones in the system do have such a result; so the search should be forwarded to other zones. Fortunately the search space can be pruned at this point. As a result of the mapping scheme, only the zones that lie to the upper left of the target point can have a result containing the given range. So if the result is not found locally, the query is forwarded to the left and top neighbors that may contain a potential result. Those nodes also check their local results and may forward the query to some of their neighbors in a recursive manner.

Figure 9 shows how a query can be forwarded in the system. If the range query cannot be answered at its target zone, *zone-10*, then it is forwarded to *zone-3* and *zone-5* which may have a result for the query. Note that forwarding is only used if the query cannot be answered at the target zone.

Forwarding is similar to flooding and has to be stopped at some point. For this purpose, a parameter called Forwarding Limit is used. Forwarding Limit is a real value between $0.0 - 1.0$ and used to determine how far the forwarding will go on. If it is set to 0.0, then only the target zone of the range query is checked and the query is not forwarded to any neighbor. If, on the other hand, it is set to 1.0, then the query is forwarded to all zones that are likely to have a result for the query; i.e., all the zones which have some point that lies on the upper left of the target point of the query.

For a zone $z : (\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle)$, every zone $z^{'} : (\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle)$ is a *diagonal zone* for z if and only if $a_2 \leq x_1$ and $b_1 \geq y_2$. In Figure 9, *zone-8* and *zone-4* are the diagonal zones for *zone-10*. However *zone-3* has no diagonal zone. It is obvious that a zone cannot have a diagonal zone if it lies on the left or top boundary of the virtual space. It is also possible that a zone may have no diagonal zone even if it has many zones to the upper left of itself. Figure 10 shows such a case where the *zone-6* at the bottom right corner has no diagonal zones.

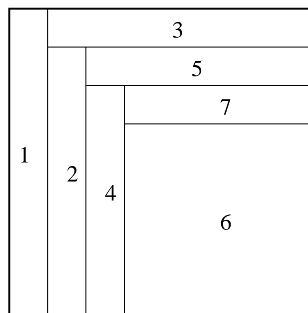

Figure 10: No Diagonal Zones

Diagonal zones are of particular interest since they are guaranteed to contain a

range including the desired answer set. This is the case because every point in the diagonal zone contains the query range and every zone in the system has at least one stored result. As the number of zones in the system increases, the possibility of finding a diagonal zone for a zone also increases.

4.4 Discussion

An important routing improvement is *Lookup During Routing*. Since the requesting zone and the target can be at any position with respect to each other (they can actually be the same zone), it is possible that a zone on the path from the requesting zone to the target zone may already have a result containing the query. The system can be improved so that every zone on the route checks its local results if it has a possible result. If the result is found, then the query is not routed any further and the result is returned to the requesting node. This way, some routing and forwarding can be avoided. The routing path decisions can be changed so that the routed queries follow a path that may have zones with possible results. The effect of *Lookup During Routing* on the system performance is shown in Section 5.3. Some of the improvements for CAN[9] are also applicable for our approach. Multiple realities, better routing metrics, overloaded zone, topology-sensitive partitioning can be incorporated into the system.

Although our system is designed for answering range queries, it can also answer exact queries. Exact match queries can be answered by setting the start and end points of the range to the exact value and then querying the system with this range. For example, in order to search for the tuples with range attribute A=20, the system is queried for the range $\langle 20, 20 \rangle$.

Updates of tuples can be incorporated into the system in the following manner. When a tuple t with range attribute $A = k$ is updated, an update message is sent to the target zone of the range $\langle k, k \rangle$. Since tuple t is included in all the ranges $\langle a, b \rangle$ such that $a \leq k$ and $b \geq k$, the update message is forwarded to all zones that lie on the upper left of the target zone. Each zone receiving an update message, updates the corresponding tuple in the local results accordingly. All zones that contain the tuple t will receive the update message and hence will update the tuple value in the stored data partition.

5 Experimental Results

We implemented a simulator in Java and then tested various aspects of our design. In this section, we present the test results. All experiments were performed on a Celeron MMX PC with 466MHz CPU and 132MBytes of main memory, running Linux RedHat 7.3.

In the experiments, a zone splits when the number of stored results exceeds a threshold value, which is called the *split point*. In the experiments, following default values are used:

- The system is initially empty. (There is only one zone in the system)
- Split point is 5, i.e., at most 5 range partitions are maintained per peer.
- Range queries are distributed uniformly.
- Domain of the range attribute is $\langle 0, 500 \rangle$.

5.1 System Performance

The performance of the system can be determined in terms of the ratio of range queries that are answered using prior answers stored in the system. Figure 11 shows the percentage of the answered range queries as a function of the forwarding limit for three different sets of queries with sizes 100, 1000, and 10,000.

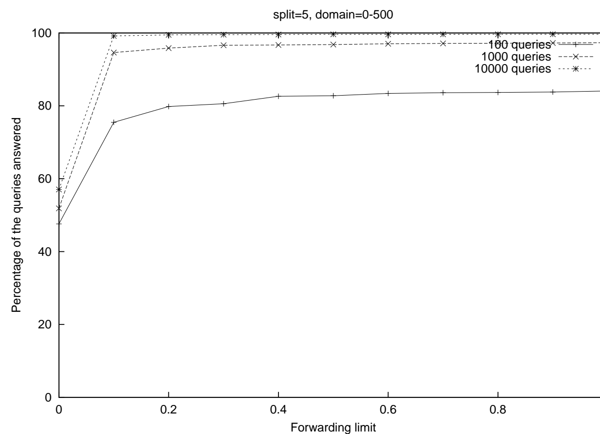

Figure 11: Effect of *forwarding limit* on the system performance

When the forwarding limit is zero, no forwarding is used and only the target zone is checked for each query. In this case, nearly half of the queries are answered and the performance improves slightly as the number of queries increases. When forwarding is enabled (even if it is set to a small value such as 0.1), there is a great improvement on the performance. With 100 queries, changing the forwarding limit from 0 to 0.1 improves the performance from 47.6% to 75.5%, whereas the performance changes from 57% to 99.2% under the same conditions with 10,000 queries.

If the forwarding limit is set to 1, then every zone that may have a possible result for the query is searched and a stored result that contains the query range is found if there exists any. When forwarding limit is set to 1 for 10,000 queries, 99.65% of the queries are answered using cached results.

We can make two important observations from Figure 11:

- *The probability of finding answers to range queries improves as the forwarding limit is increased.* This is quite clear since increasing the forwarding limit results in the search of more candidate zones.

- *The probability of finding answers to range queries improves as the number of queries increases.* As the number of queries is increased, more results are stored in the system and the possibility of finding a result for a query gets higher.

5.2 Routing Performance

The routing performance is measured in terms of the average number of zones visited for answering a query. When the forwarding limit is 0, the result is the average number of zones visited during routing. If forwarding is enabled, it also includes the zones visited during forwarding in addition to those visited during routing.

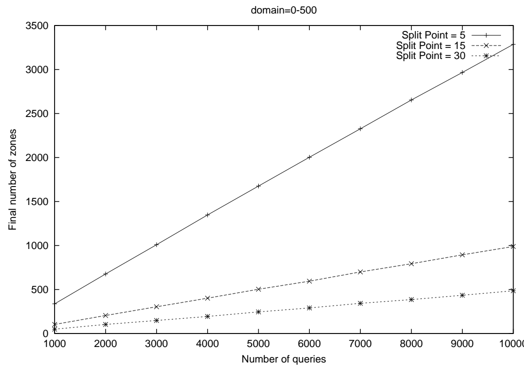

Figure 12: Number of final zones as a function of *split point*

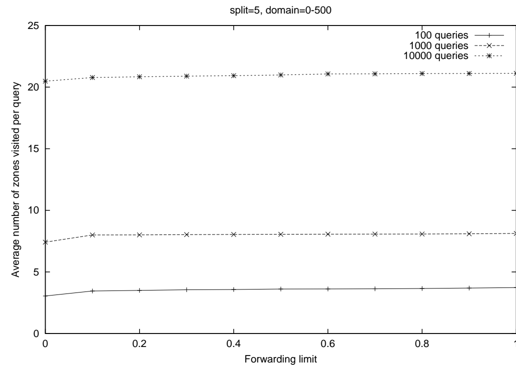

Figure 13: Effect of *forwarding limit* on the number of visited zones

Figure 12 shows that the final number of zones in the system increases linearly with respect to the number of queries. In Figure 13, we vary the forwarding limit and notice that the average path length is around 8 zones for 1000 queries and 21 for 10000 queries. Even though the final number of zones in the second case is 10 times that in the first case, the path length increase is less than 3 times.

In comparison to the reported performance of CAN [9], we note that the average number of hops is less using our approach because the system is initially empty and so very few zones are visited for earlier queries. As more queries are answered, the number of zones in the system increases as well as the number of zones visited for the query. In Section 5.5 we examine the number of visited zones when the system is not empty initially.

As seen from the Figure 13, changing the value of the forwarding limit in the interval 0.1-1.0 does not change the result too much, so we can conclude that most of the results are found in nearby neighbors during forwarding.

In the above discussion, the *forwarding limit* determined the portion of the entire coordinate space that needs to be searched. In addition to the above mentioned forwarding, we tried to restrict the number of zones visited during forwarding in a manner where the *forwarding limit* determines the forwarding space in proportion to the space in left and up of the target point in the coordinate space. Our experiments with restricted forwarding had similar results as the normal forwarding.

5.3 Lookup During Forwarding

One of the improvements to the system is to implement *Lookup During Routing* so that the results for the queries may be found while they are being routed to their target zone. If a result for the query is found on its way to its target zone, the query is not routed anymore which results in less number of visited zones. Figure 14 shows the number of visited zones per query when *Lookup During Routing* is used. The percentage of the queries that are answered during routing is shown in Figure 15.

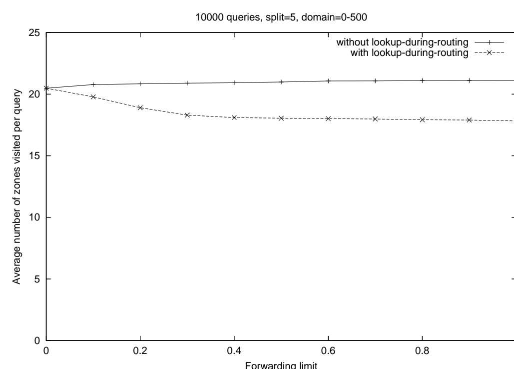

Figure 14: Effect of *Lookup During Routing* on the system performance

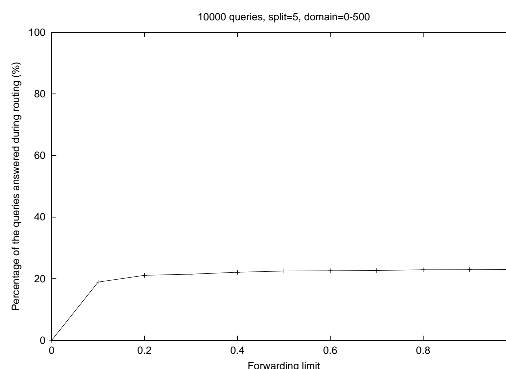

Figure 15: Percentage of the queries answered during routing

5.4 Selectivity

Fig 16 shows the performance of the system when query ranges are restricted to certain maximum lengths. The domain of the range attribute is changed to 0-10000 in order to avoid the repetitions of queries. In the figure, *Selectivity* $k\%$ means that the length of any queried range is less than or equal to $(k * |domain|/100)$ where $|domain|$ is the length of the domain and equals 10000 in this case. For example, with 0.1% selectivity, all query ranges have length less than or equal to 10 since $0.1 * 10000/100 = 10$. 100% selectivity is the same as no selectivity since the query ranges can have any possible length. When creating the range queries, the start points of the ranges are selected uniformly from the domain of the range attribute and then the length of the range is determined randomly according to the selectivity.

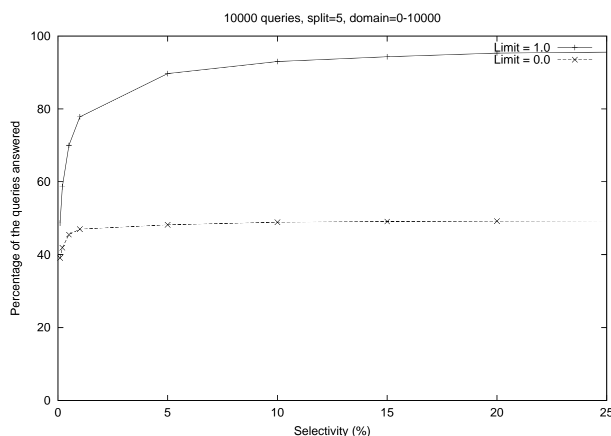

Figure 16: Effect of *selectivity* on the system performance

As seen from the figure, the percentage of queries answered decreases as the selectivity gets smaller. That is because restricting the query ranges to a smaller length makes it harder to find prior results that contain a given range. When the selectivity is small, a query is looking for a very specific range. All the prior queries have also been quite specific. Hence the probability that the current query intersects or is contained in one of the previous queries is low, which explains the observed behavior.

5.5 Cold Start vs. Warm Start

Although in most of our experiments the system is initially empty, it is also useful to test the performance and the path length when the system is not initially empty

but has some number of initial zones. This means that there are some results stored in the system prior to answering queries and the number of these initial results is proportional to the number of initial zones.

Figure 17 shows the performance of the system when there are different number of initial zones. It is seen from the figure that increasing the number of initial zones in the system slightly increases the performance. On the other hand, the average path length increases because now there are more zones in the system and even the initial queries require many zone visits. Figure 18 shows the number of visited zones per query when there are initial zones in the system.

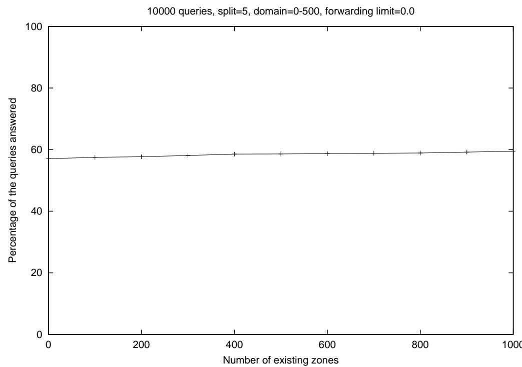

Figure 17: Effect of *warm start* on the system performance

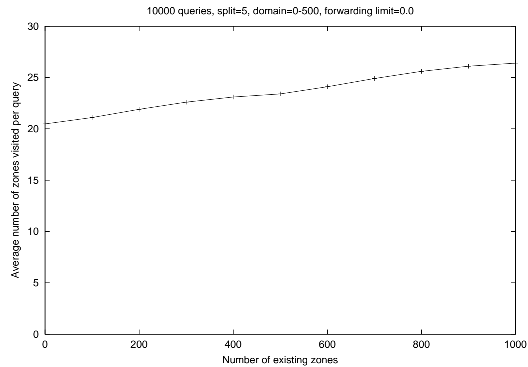

Figure 18: Effect of *warm start* on the average path length

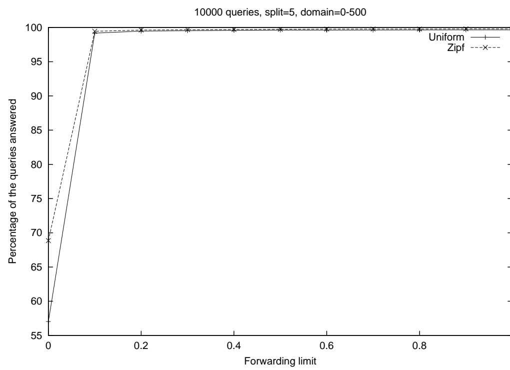

Figure 19: Percentage of answered queries for different distributions of the range queries

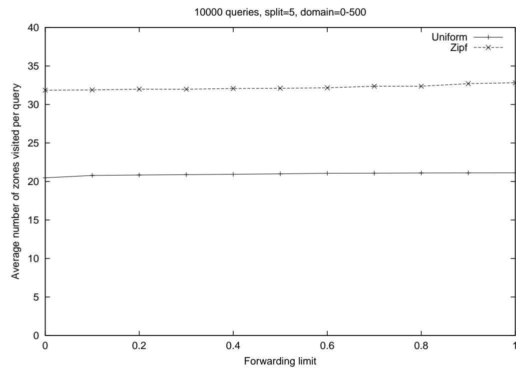

Figure 20: Average number of hops visited for different distributions of the range queries

5.6 Uniform vs. Zipfian Query Distribution

We tested our approach using query sets that have a zipfian distribution. The query set is created as follows: The start points of the ranges are selected according to zipf distribution so that they are clustered near 0. After the start point of the query is determined, the end point is selected uniformly from the remaining interval (start point, 500] since 500 is the maximum possible value of the range attribute in this case.

Figure 19 shows the performance for different distributions of the query ranges. With zipfian distribution, the queries are mapped to a certain region (in our case, they are mapped to points that are close to the y axis since the start points of the ranges are usually close to 0) and there is a better chance of finding similar results in the system. However the average number of visited zones is much greater when the query ranges have a zipfian distribution rather than a uniform distribution as shown in figure 20. When queries have a zipfian distribution, they are usually hashed to certain portion of the virtual space and that portion is divided into many small zones. Since further queries are also hashed to the same portion with a high probability, the queries visit these small zones during routing and forwarding, which results in a much greater path length.

6 Conclusions and Future Work

Peer-to-peer systems are gaining in importance as they distribute and connect users and information that are distributed across the globe. The true distributed systems of today need to facilitate this world wide retrieval and distribution of data. So far most peer-to-peer attempts have been restricted to exact match lookups and therefore are only suitable for file-based or object-based applications. This paper represents a first step toward the support of a more diverse and richer set of queries. Databases are a natural repository of data, and our enhanced CAN based system supports the basic range (or selection-based) operation. Our approach is simple and very promising. We have shown how to exploit the CAN approach to support range queries and have demonstrated that it successfully scales using a variety of performance studies. This paper represents a first step toward the design of a complete peer-to-peer database system. In the short term, however, we plan to explore extending our approach to support multi-attribute range queries as well as non-integer based domains.

References

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

[2] Steven Gribble, Alon Halevy, Zachary Ives, Maya Rodrig, and Dan 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.

[3] Abhishek Gupta, Divyakant Agrawal, and Amr El Abbadi. Approximate range selection queries in peer-to-peer systems. Technical Report UCSB/CSD-2002-23, University of California at Santa Barbara, 2002.

[4] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Than Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

[5] Matthew Harren, Joseph M. Hellerstein, Ryan Huebsch, Boon Than Loo, Scott Shenker, and Ion Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

[6] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views (extended abstract). In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 95–104. ACM Press, 1995.

[7] Qin Lv, Sylvia Ratnasamy, and Scott Shenker. Can heterogeneity make gnutella scalable? In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, 2002.

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

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott 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.

[10] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari 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.

[11] Y. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, 2001.