An Adaptive Protocol for Efficient Support of Range Queries in DHT-based Systems

Jun Gao[1] Peter Steenkiste[1,2]
[1]School of Computer Science [2]Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{jungao,prs}@cs.cmu.edu

Abstract

In recent years, Distributed Hash Tables (DHTs) have been proposed as a fundamental building block for large scale distributed applications. Important functionalities such as searching have been added to the DHT's basic lookup capability. However, supporting range queries efficiently remains a difficult problem. In this paper, we describe an adaptive mechanism that relies on a logical tree data structure, the Range Search Tree (RST), to support range queries efficiently. Nodes in the RST automatically group registrations based on their values. Queries are decomposed into a small number of sub-queries for efficient resolution. The system dynamically optimizes itself to minimize the registration and query cost based on observed load. The system is fully distributed and avoids bottleneck problems encountered in traditional tree-based systems. Extensive simulation results validate the effectiveness of the system.

1. Introduction

Overlay networks based on Distributed Hash Tables (DHTs) [1] provide a scalable and robust data location and lookup substrate for large scale distributed applications and services, including wide area file and storage systems [6, 20], content/service discovery systems [4, 9], information retrieval systems [18, 23], and distributed databases [13]. A DHT supports exact name lookup only: a publisher registers a data item with a node in the system using an identifier that is typically the hash of the name of the data item, e.g., the name of a file. When searching for the data item, a client uses the "name" of the item to generate the same identifier by applying the hash function. As an example, consider a nationwide traffic monitoring service that is built on top of a DHT. Devices such as cameras and sensors are installed along the road side or mounted on patrol cars to monitor traffic status and road conditions. Users of such a service may pose a variety of queries, e.g., "*return the observed speed at I-70 Exit 6*".

Recently several groups [4, 18, 9] have studied how to use DHTs to support searching based on subset matching. In particular, descriptive names defined as a set of attribute value pairs (AV-pairs) are used to represent content, and the system returns any name that matches the AV-pairs specified in a query. The basic idea is to register a name at multiple nodes according to its AV-pairs, and a query that is the subset of the name may be resolved by visiting these nodes. While DHTs are efficient for **point queries** that require an exact match, they perform poorly for **range queries** [11, 13], where a range rather than a specific value is specified for an attribute.

Range queries are common and important for discovery and exploration purposes, as a user may not know exactly what he is looking for. For example, a driver may issue the query "*return the speed observed by cameras that are between Exit 10 and Exit 50* ($\texttt{10} \leq \texttt{exit} \leq \texttt{50}$)", so that he may choose to get off the highway early to avoid congestion down the road. A police patrolling a highway section with speed limit of 55 mph may ask the system to "*return the list of cameras that observe speeds higher than 75 mph* ($\texttt{speed} \geq \texttt{75}$)". A naive way to resolve a range query is to issue separate point queries to nodes that correspond to each possible value within the query range. For large range queries, which are typical for exploration purposes, this becomes expensive.

In this paper, we propose an adaptive range search mechanism that utilizes a logical tree data structure, the Range Search Tree (RST). Each level of the RST corresponds to a different data partitioning granularity. Registrations may be aggregated at different levels. Range queries are decomposed into $O(\log R_q)$ sub-queries, where R_q is the range length, and they are resolved by nodes that correspond to each sub-query. The RST is implicit in that nodes are only "filled in" as needed. Our system is self-tuning: it optimizes itself based on the type of queries and load it observes to achieve efficiency for both queries and registrations. The system works in a fully distributed fashion since all decisions are made locally.

The rest of the paper is organized as follows. Section 2 provides an overview of the system. In Section 3, we describe a set of algorithms that use a static RST to facilitate range queries. This serves as a basis for the adaptive algorithms described in Section 4 that further enhance the system's efficiency. Section 5 presents the results of our simulation study. We discuss related work in Section 6 and conclude in Section 7.

2. System Overview

We describe the challenges associated with supporting range searches in the context of a DHT-based content discovery system (CDS).

2.1. Content Discovery

A CDS [9] is a distributed system that enables content discovery. A piece of content is represented by a descriptive *content name*, which consists of a set of AV-pairs in the form of $CN : \{a_1 = v_1, ..., a_n = v_n\}$. The specific meaning of a content name depends on the application. In the traffic monitoring service, it is used to represent a camera, and may have attributes such as `speed, location`, and `view`. Queries also consist of AV-pairs. We differentiate two types of queries: point queries and range queries. In a point query, all AV-pairs are equality predicates, while in a range query, some of the AV-pairs are inequality predicates.

A CDS can be built on top of a DHT such as Chord [22]. In a DHT, each node is assigned a unique numerical node ID in an m-bit key space. The node ID serves as its overlay network address. Messages are delivered in the system based on keys, which are typically generated by hashing a data item. In our system, a content name is registered with each node whose ID is numerically closest to the hash of one AV-pair in the name. When a node receives a registration, it inserts the name into its local database. To resolve a point query, the system applies the hash function to one of the query's AV-pairs, and sends the query to the corresponding node. When a node receives a query, it compares each AV-pair in the query against the AV-pairs in each name in its database, and returns the set of names that match the query. A content name matches a query if it satisfies all the AV-pairs in the query simultaneously.

It is worth noting that the CDS supports dynamic content, which must be periodically refreshed and updated. Refresh messages are also important as a fault tolerance mechanism: if a node that is responsible for a content name leaves or crashes, the update messages will store the name at a live node whose ID is now closest to the hash. Before we discuss issues related to range queries, we first explain how a CDS can handle load balancing, which is critical to the system's performance.

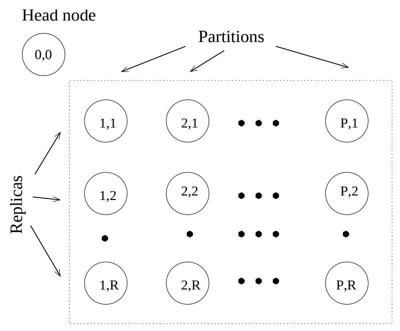


Figure 1. Load balancing matrix for a data item.

2.2. Load Balancing

In real applications, the distribution of AV-pairs is often skewed. For example, some AV-pairs are common in many content names or queries. As such, nodes responsible for these AV-pairs will be overloaded by registrations or queries, and become "hot spots", i.e., bottlenecks of the system. Without proper load balancing, the system's performance will degrade quickly. We briefly describe the particular load balancing mechanism we use. More details can be found in [9].

The idea is that when a node corresponding to a data item, e.g., an AV-pair, gets overloaded by registrations or queries, the system uses, instead of one node, a set of nodes that are already in the system to share the load. This set of nodes is organized into a logical matrix, termed a *load balancing matrix* (LBM). Each column of the matrix holds a *partition*, i.e., a subset of the content names that contain this AV-pair (which helps spread registration load), and nodes in different rows within a column are replicas of each other (which helps spread query load). Figure 1 shows the layout of the matrix for a data item. Each node in the matrix has a column and row index, (p, r), and its node ID corresponds to the hash of the data item (AV-pair in this case), the column, and row indices together:

$$N^{(p,r)} \leftarrow \mathcal{H}(data\_item, p, r).$$

To register a data item or issue a query, an endpoint must know the size of the corresponding LBM. It may retrieve the size from the *head node* (ID corresponding to $\mathcal{H}(data, 0, 0)$), which stores the matrix's current size. The size can also be obtained by directly probing nodes in the matrix to avoid overloading the head node. For registration, the endpoint selects a random partition from the matrix and sends the item to each node in that partition. To retrieve all the possible matches, the query is sent to a random node in each partition. The random distribution of registrations and queries ensures that with high probability, the load is spread evenly among all nodes in a matrix.

Load balancing, or matrix expansion/shrinking, is done in a distributed fashion. The LBM starts with 1 node, and if the registration load exceeds a threshold maintained on this node, T_{reg}, it will issue a request to the head node to increase the matrix size, e.g., from 1 to 2. Future registrations will then be shared by these two partitions. If the registration load further increases, more partitions will be included in the LBM. The number of partitions, P, is proportional to the registration load: $P = \lceil \frac{L^R}{C_R} \rceil$, where L^R is this data item's registration load, and C_R is the capacity of each node. Similarly, the number of replicas in the LBM is proportional to the query load for this data item.

2.3. The Range Query Problem

In a range query, at least one attribute is specified by a range instead of a single value. If the query also contains AV-pairs with equality predicates, we may choose one of them for resolving the query, and the inequality comparison is done at the corresponding node. This way we essentially treat the range query as a point query. However, this may not always be applicable, for example, when all AV-pairs in the query contain ranges. In addition, the AV-pairs with equality predicates may be popular and have many partitions, and we may not want to query that matrix for performance reason. In this paper, we focus on the scenario where an AV-pair with a range is used for query resolution.

Depending on how registration is done, there are two straightforward ways of supporting range queries. First, we still apply the hash function to the attribute and value together as we did before. This is efficient for point queries, but to resolve a range query, $Q : \{s \leq a \leq e\}$, or $Q : [s, e]$ for short, it must be broken up into $R_q = e - s + 1$ sub-queries, and sent to each node that corresponds to a value in the range. This approach works well if registrations are dense and queries cover relatively small ranges. However, it is an $O(R_q)$ approach and the number of query messages will become prohibitive when the query range increases. Moreover, if the registrations within this range are sparse, most of the query messages will be wasted.

Alternatively, we can apply the hash function to the attribute only. This way all the content names that share the same attribute will end up registering at the same node, irrespective of the value; at query time, all point queries and range queries will also be sent to the same node for resolution. This approach performs well under light load, in that no matter what the range size is, all queries will have the same overhead. However, this node will become overloaded as the load increases. Fortunately, the load balancing mechanism will help by using more nodes to distribute the load. But this solution is not efficient: each partition contains registrations with random values, so every query, including point queries, will have to visit all the partitions.

We observe that both approaches work well in some cases, but perform poorly in other cases. The problem is that neither solution takes into consideration the range of queries and the registration and query load. An ideal solution would behave similarly to the first approach for attributes that experience mostly point queries or queries over small ranges, but it would behave similarly to the second approach for attributes that experience light registration load and large query ranges. Before we present a system that exhibits this adaptive behavior (Section 4), we first describe a static tree-based approach.

3. Static Range Query Mechanisms

Our design to support range search is based on the range search tree (RST) data structure. In this section we introduce the RST organization and describe its use.

3.1. Range Search Tree (RST)

Assume we have an attribute a that takes on numerical values and may be searched using range queries. Suppose the domain of a is D_a, and D_a is bounded; values can be continuous or discrete. We break up D_a into n sub-ranges and represent each sub-range by its lower bound. D_a is thus the union of the sub-ranges $\{v_0, v_1, ..., v_{n-1}\}$, where $v_i < v_j$, if $i < j$. For example, if the attribute is `speed` in mph, we could break up the speed range into sub-ranges of 5 mph, and the value 35 would represent the range $35 \leq$`speed`$<$ 40. Note that the sub-ranges may not be equal sized if we have prior knowledge of the distribution of the values of a.

The RST is a complete and balanced binary tree with n leaf nodes and $\lceil \log n \rceil + 1$ levels. (We assume n is a power of 2; otherwise, we round it up to the next power of 2 by filling in extra values.) Levels are labeled consecutively with the leaf level being level 0. Each node in the tree represents a different range. Leaf nodes correspond to the smallest sub-ranges, and each non-leaf node corresponds to the union of its two children. At level l, the range of the ith node from the left represents the range $[v_i, v_{i+2^l-1}]$. The union of all ranges at each level covers the full domain. The RST structure is similar to the segment tree data structure [21] used in computational geometry and spatial databases. A special case of an RST is a "unit RST" in which the domain is integer numbers and each leaf node represents one integer value. Figure 2(a) is an example unit RST with domain

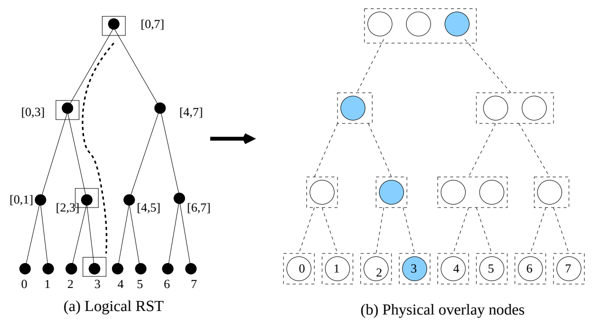


Figure 2. (a) A logical RST. The dotted curve illustrates $Path(3)$. (b) Overlay network nodes this RST is mapped onto. A circle represents a physical node and a dotted rectangle represents an LBM. Filled nodes are selected by the registration algorithm to receive $\{a = 3\}$.

$[0, 7]$. In the rest of the paper, we will present algorithms for a unit RST, but they generalize easily to a general RST.

We assume that each attribute's domain is known to all nodes in the system. As such, the logical structure of an attribute's corresponding RST is also known in the system. Each node in the RST can be mapped onto a set of physical nodes, or an LBM in the overlay network (Figure 2(b). Given a range $[s, e]$, the node IDs in the overlay network that this range is mapped onto correspond to the hash of the following 4-tuple: the attribute, the range, the column and row indices in its LBM. For example, in Figure 2(b), the node ID of the filled node in the root level corresponds to $\mathcal{H}(a, [0, 7], p = 3, r = 1)$. The LBMs may have different number of partitions and replicas depending on the load they receive. In Figure 2(b), for each LBM, we only show 1 replica for clarity. It is important to note that the parent-child relationships between nodes do not need to be actively maintained, and a node can infer its parent or children's range based on its own range.

3.2. Registration

We now describe how an AV-pair $\{a = v\}$ is registered using RST. We observe that in the logical RST, there is exactly one node whose range covers the value v at each level, and this set of nodes forms a path from the leaf node $N[v, v]$ to the root. We name it $Path(v)$. In the static RST design, we register $\{a = v\}$ with each LBM corresponding to each node in $Path(v)$. Figure 2 illustrates the registration of $\{a = 3\}$: it is registered with each physical node within a selected partition of each level's LBM. This registration algorithm automatically aggregates AV-pairs at different granularity. As the level number increases, since there are fewer nodes in the RST, the LBM corresponding to one RST node may have more partitions. For example, in Figure 2(b), each LBM at the leaf level has one partition, and the root level LBM consists of 3 partitions.

Since the structure of an RST is determined by the domain of the corresponding attribute, registrations are carried out in a fully distributed fashion: based on the value in an AV-pair, an endpoint can locally determine the set of nodes in the network that it should register with and does not need to consult with any other node or traverse the tree. As a result, no bottlenecks are created in the system.

3.3. Query

Given the registration mechanism, there are many ways to decompose and resolve a range query using the RST. The efficiency of a query algorithm is determined by how the range is decomposed. To find an efficient algorithm, we introduce the *relevance* metric to guide our design. Formally, suppose a query algorithm decomposes a query $Q : [s, e]$ into k sub-queries, corresponding to k nodes in the RST, $N_1, ..., N_k$. The relevance r of this algorithm is defined as:

$$r = \frac{R_q}{\sum_{i=1}^{k} R_i},$$

where R_i is node N_i's range length, and R_q is the query's length. Clearly, $0 < r \leq 1$.

Intuitively, the relevance indicates how well the query range matches the RST nodes that are being queried. Low relevance algorithms, such as sending a point query to the root level, may be inefficient for both queries (visit nodes with a low concentration of relevant registrations) and registrations (the query load concentration may cause the root level to replicate often). In contrast, decomposing a query to leaf level nodes has a relevance of 1. From the registration's point of view, there will be no unnecessary replications. From the query's point of view, this may require too many sub-queries, so it is not a desirable algorithm either.

We design a query algorithm that maximizes the relevance ($r = 1$) (to minimize the registration cost) while using a small number of sub-queries (to minimize the query cost). The algorithm is based on the following theorem:

Decomposition Theorem. *A query $Q : [s, e]$ with length R_q can always be decomposed into $O(\log R_q)$ disjoint subranges, each corresponding to the exact range of a node in the RST.*

The proof can be found in [8]. We call this set of nodes the minimum cover (MC) of range $[s, e]$. The level of the highest node in the MC is $\lfloor \log R_q \rfloor$.

The query algorithm works as follows. To resolve query $Q : [s, e]$, the querying node locally determines the MC for range $[s, e]$ by running a simple top-down recursive algorithm that first finds the RST node that has the largest range within $[s, e]$ (the highest node in the MC) and then recursively repeats this process for the segments of the range

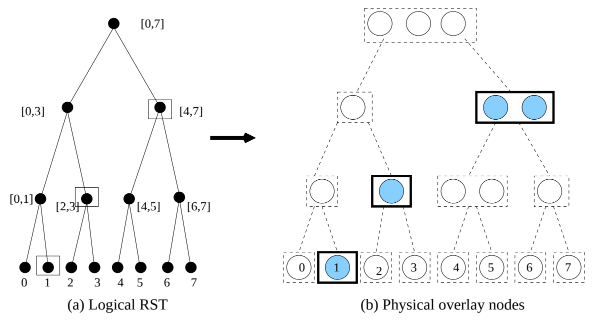


Figure 3. Range [1, 7] is decomposed into 3 sub-ranges indicated by the nodes with a box. Filled nodes will receive the query.

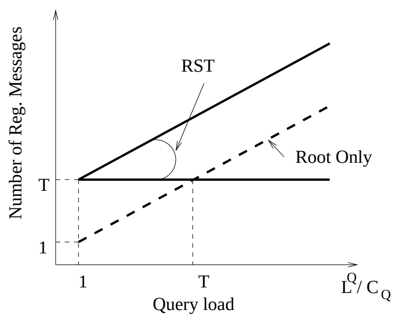


Figure 4. Number of registration messages needed for $\{a = v\}$ as query load increases.

not yet decomposed. Once the MC is computed, the querying node then sends the query to the LBMs in the overlay network that correspond to each MC node. Figure 3 is a decomposition example for query $Q : [1, 7]$.

The query algorithm is also fully distributed and deterministic, in that the decomposition is done by the querying node itself based on its query range, and no traversal of the tree is needed. The algorithm separates queries with different ranges, thus avoiding bottlenecks. For example, a point query will be sent to the leaf level, and a large query will use nodes higher up in the tree.

3.4. Cost Analysis

We now analyze the cost of registrations and queries using the static RST. We assume that the domain for attribute a is $D_a = [0, n-1]$ and its RST has $T = \lceil \log n \rceil + 1$ levels. We first consider registration cost.

From Section 3.2, we know that the number of registration messages needed to register pair $\{a = v\}$, N_R, is determined by the height of the RST and the query load that comes to this RST. More specifically, it equals the total number of replicas (rows) in the LBMs that are on $Path(v)$. Assume L^Q is the total query load for attribute a. If the query load on the node at level t in $Path(v)$ is L_t^Q, then the number of replicas in its LBM is $\lceil \frac{L_t^Q}{C_Q} \rceil$, with C_Q being the maximum query capacity of a node, e.g., the query rate that this node can sustain. Thus,

$$N_R = \sum_{t=1}^{T} \lceil \frac{L_t^Q}{C_Q} \rceil.$$

The value of N_R has the following bounds (Proof in [8]):

$$T \leq N_R \leq T + \lceil \frac{L^Q}{C_Q} \rceil.$$

The maximum occurs when the value v is contained in every query range in L^Q. The minimum T corresponds to the case that there are no replicas on $Path(v)$. Figure 4 illustrates the bounds: N_R lies between the two solid lines.

We compare the cost of using a static RST with the approach where we only use the root. In that case, the number of messages needed for *any* registration, N'_R, is determined by the number of replicas in the root level LBM: $N'_R = \lceil \frac{L^Q}{C_Q} \rceil$. The cost grows linearly with the query load, as is shown by the dotted line in Figure 4. The Root Only case is more efficient when the query load is low, specially, when the number of replicas at the root level is smaller than the height of the RST, i.e., $L^Q/C_Q \leq T$. As the load increases beyond that point, the registration cost using RST is generally lower than using only the root.

The number of query messages N_Q needed to resolve a query $Q : [s, e]$ equals the total number of partitions in the LBMs that correspond to the nodes in Q's MC. N_Q has a similar bound [8]. It is at least $O(\log R_q)$, if each LBM of an MC node has only 1 partition, and could be higher if some of the LBMs have more than 1 partition. In Figure 3, $N_Q = 4$, as query $Q : [1, 7]$ will be sent to the 4 filled nodes. In comparison to using only the root level, the static algorithm wins when the registration load is high and the root level LBM has more partitions than $O(\log R_q)$. When the load is low, using the root level without further decomposition is more efficient.

The above analysis points out the following deficiencies in the static algorithms: (1) Proactively registering with all levels of the RST without considering query ranges can be wasteful. For example, if all query ranges are smaller than R_q, then registering with levels higher than $\lfloor \log R_q \rfloor$ is unnecessary, since no query will be sent there. (2) Decomposing a range solely based on its length while ignoring the registration and query load information is suboptimal. If both the registration and query load in a subtree of the RST are low, we do not need to decompose the query in the subtree and should just use the subtree root. Subsequently, we do not need to register with lower levels in the subtree.

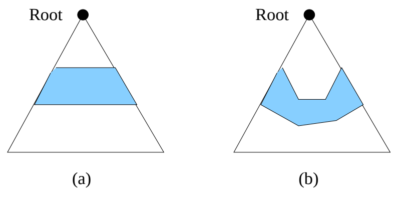


Figure 5. RST with (a) a flat band, and (b) a ragged band.

4. Dynamic Range Query Mechanisms

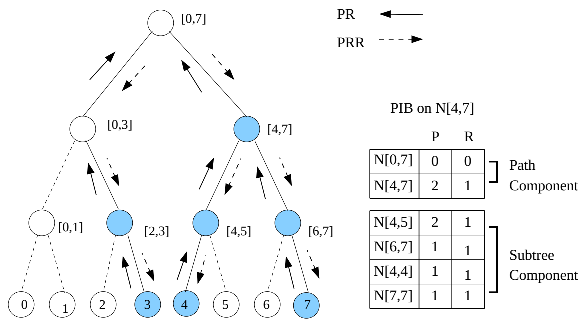
We now present a dynamic range query design. While in the static mechanisms, registrations go to every level of the RST regardless of the queries, the idea here is to only register with the nodes that are needed based on the query ranges and the load information. We call this set of nodes the *band* (Figure 5): only nodes in the band will accept registrations and be able to resolve queries. As such, only the LBMs corresponding to the band nodes will have a non-zero size. The shape of the band is not necessarily flat (Figure 5(b)), and it changes depending on the queries. For example, if the query load is low and the query ranges are large, the band will migrate upwards toward the root.

In this section, we first present the Path Maintenance Protocol, which allows endpoints to discover the band for registration and query. We then present endpoint algorithms to show how registrations and queries are carried out with a dynamic band. Finally, we present the local algorithms executed on nodes to adapt the band.

4.1. The Path Maintenance Protocol

The goal of the Path Maintenance Protocol (PMP) is to propagate information to nodes in the RST, so that endpoints can learn about the band for registrations and queries. We call this information the Path Information Base or PIB. Recall that for load balancing purpose, the size of an LBM is stored in its head node. For range queries, the head node also maintains the matrix sizes of some other nodes in the tree. The PIB consists of two components. The *path component* contains the matrix size of nodes in the path from this node to the root of the RST, and the *subtree component* contains the matrix size of nodes in this node's subtree.

The PIB is established through the exchange of Path Refreshing (PR) and Path Refreshing Reply (PRR) messages (Figure 6). Each head node in the band periodically sends a PR message to its parent; the message contains the node's current subtree component of its PIB. When a node receives a PR message, it updates the corresponding entries in its PIB, and then sends a new PR message to its parent. The PR message will eventually reach the root. Through the PR messages, each node collects up-to-date subtree status. The



	P	R
N[0,7]	0	0
N[4,7]	2	1
N[4,5]	2	1
N[6,7]	1	1
N[4,4]	1	1
N[7,7]	1	1

Figure 6. PMP message exchange among head nodes. Filled nodes are in the band. Matrix sizes are from Figure 2(b).

periodical PRR messages are initiated by the root and traverse down the tree along the reverse paths of the PR messages. In particular, a node sends a PRR message to each of its children from which it receives a PR message before. The PRR message includes the path component of its PIB. Upon receiving a PRR message, a node updates its PIB's path component, and then sends its own PRR message.

In addition to allowing nodes to establish their PIBs, the PMP messages also act as a mechanism to re-establish the PIB if nodes fail or leave the system. For example, if a parent node fails, the PR messages from its children will be forwarded to a proper live node by the DHT layer, which will become the new parent.

4.2. Endpoint Algorithms

Using the PIB, endpoints can issue registrations and queries in a fully distributed fashion.

4.2.1. Registration

To register $\{a = v\}$, an endpoint first discovers the band by probing any head node in $Path(v)$. The band discovery step is unnecessary in the static mechanisms, since the band is the full tree. Nodes in lower levels are preferred when probing, since collisions with probes for other values are less likely to occur at those levels. If the node being probed has an established PIB, it can provide the matrix size information for all nodes in $Path(v)$ to the probing endpoint. The endpoint then registers only with the nodes that are in the band. Endpoints may cache the retrieved path information for future use, thus avoiding repeated probing.

If the node being probed does not have a PIB because it is too low in the tree and the PMP messages did not reach it, the node replies with a NULL. The endpoint can then probe a node higher in the path, e.g., the node located halfway between the first probed node and the root in $Path(v)$.

The first registration is a special case, because all the probes will return NULL. To handle this, we define a default band, whose level is known to the endpoints. The registration will go to the default level, and the head node of the default band will then start the PMP message exchanges. One option is to use the root level as the default band.

4.2.2. Query

To resolve a query $Q : [s, e]$ with length R_q, the querying node also needs to retrieve the band information first. Clearly, probing the root node suffices since its PIB contains the whole tree. However, the root node may become a bottleneck if all probes go there. It is shown in [8] that for Q, there exist at most 3 adjacent RST nodes at level $\lfloor \log R_q \rfloor$ that cover the query. We choose to probe these nodes to minimize collisions with probes from queries with different lengths and ranges. Each probed node returns its complete PIB to the querying node.

Based on the returned PIB information, the querying node reconstructs the logical RST and annotates each RST node with its LBM's size, which reflects the load status. It then executes a local algorithm to decompose the query range. The algorithm differs from the static decomposition in two aspects. First, while the decomposition is still done in a top-down fashion and tries to find the highest RST node that is within the query range, it only considers nodes within the band. Second, while the static algorithm does not consider the load status within the tree, the goal of the dynamic query decomposition is to minimize the total number of physical nodes that should be contacted. In particular, if a root of a subtree has fewer partitions, the algorithm will use it rather than the subtree. One exception is that if the number of replicas of the subtree root exceeds a threshold, then we will continue to decompose the query to avoid further overloading the root. Figure 7 shows the same example as in Figure 3, but at the subtree rooted at $N[0, 3]$, we use it rather than $N[0, 1]$ and $N[2, 3]$ to reduce the query cost from 4 to 3 for query range $[1, 7]$.

4.3. Distributed Band Adaptation

The band allows us to minimize the cost of registrations and queries for a given load. However, the load may change, and if the band does not adapt, the system may perform poorly under the new load. Consider the example in Figure 7. At a given time, due to previous point queries, suppose the band only contains nodes from the lowest level. Now new queries come in with range $[0, 3]$, and using the current band would require breaking up each query into 4 small sub-queries. This is clearly inefficient comparing with the scenario where node $N[0, 3]$ is in the band. In that case, only 1 query message is needed, since $N[0, 3]$ corresponds to 1 partition in the overlay network.

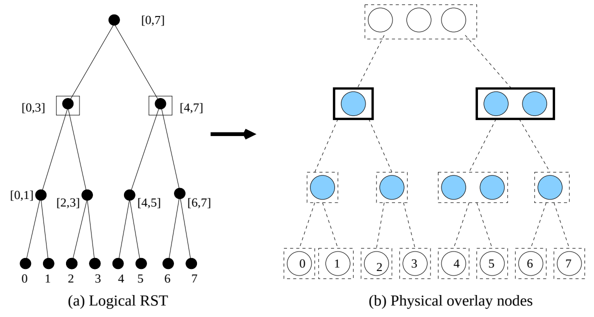


Figure 7. Query range $[1, 7]$ is decomposed into 2 sub-ranges indicated by the solid boxes. Filled nodes are in the band.

4.3.1. Adaptation Actions

Band adaptation occurs in the nodes at the top or bottom edge of the band. A node knows its position in the band from its PIB. Each such node periodically performs a cost/benefit analysis for including its parent or children, or removing itself from the band. The goal is to reduce the total number of query and registration messages observed by the system.

To make adaptation decisions, each node maintains statistics on the type of queries it receives. Consider query $Q : [s, e]$ arriving at node $N[v_1, v_2]$ ($R_N = v_2 - v_1 + 1$), and suppose the sub-query to be resolved by N is $Q_i : [s_i, e_i]$. Note that $[s_i, e_i] \subseteq [s, e]$, and $[s_i, e_i] \subseteq [v_1, v_2]$. N classifies Q into the following categories: **Large Query**, if $R_q \geq 2R_{N_i}$; **Left Partial Query**, if $[s_i, e_i] \subseteq$ range of N's left child; **Right Partial Query**, if $[s_i, e_i] \subseteq$ range of N's right child; **Middle Partial Query**, if $[s_i, e_i]$ intersects with both of N's children. A node may take the following four possible adaptation actions depending on the cost analysis.

Top Expansion (TE). A node at the top of the band periodically evaluates whether including its parent node in the band will reduce the overall cost. With the parent in the band, the cost of future Large Queries will be reduced, since they will be sent to the parent based on the query algorithm. However, this also means an increase of the registration cost, since future registrations must be sent to both the child and the parent level. When the percentage of Large Queries received by this node exceeds a threshold T_{large}, the decrease of query cost will outweigh the increase of registration cost, and the node will expand the band to include the parent by duplicating its contents at the parent. To ensure consistency, the node will request its sibling to duplicate as well. We will discuss how T_{large} is set shortly.

Bottom Expansion (BE). Similarly, a node at the bottom of the band evaluates the benefit of including its children into the band. By doing so, the cost of future partial queries will be reduced, e.g., a Left Partial Query would be sent to the left child, which typically has fewer partitions than the parent. Of course, the cost of expansion is an increase in registration cost. As a result, when the percentage of partial queries exceeds thresholds T_{left}, T_{right} or T_{middle}, the node will include its left child, right child or both into the band by sending them contents.

Top Reduction (TR). This is the reverse action of TE. A node at the top of the band may remove itself from the band to reduce future registration cost. Future queries destined to it will go to its children, and from a child node's point of view, these queries are Large Queries. Similar to TE, if the percentage of these queries becomes small, the node would leave the band by informing its corresponding matrix's head node to set the matrix size to zero. This information will be propagated to other nodes during the next round of PMP messages.

Bottom Reduction (BR). This is the reverse action of BE. A node at the bottom of the band may remove itself to reduce registration cost. Removing itself means all queries it would have received will go to the parent. Thresholds T_{left}, T_{right} and T_{middle} will be used for this decision.

4.3.2. Distributed Algorithm

Band adaptation actions are carried out in a distributed fashion, since each decision is made by a node based on its local information. To set the proper thresholds for the various actions, a node may need additional load information such as query and registration rate from its sibling or parent, but this remains a localized operation.

We use a simple example to illustrate how T_{large} is set for TE. Other thresholds can be computed similarly. Consider the subtree rooted at $N[0,1]$ in Figure 7. Suppose initially only $N[0,0]$ and $N[1,1]$ are in the band and each corresponds to 1 partition and 1 replica. $N[0,0]$ (same on $N[1,1]$) computes the total number of messages the $N[0,1]$ subtree receives within a time interval Δ as follows:

$$M = M(query) + M(reg) = (2r_q p\Delta + r_q(1-p)\Delta) + r_{reg}\Delta,$$

where r_q and r_{reg} are query and registration rates arriving at the subtree; p is the percentage of Large Queries (2 query messages for each large query and 1 for other queries); all registrations require 1 message. If TE were to take place, the total number of messages becomes:

$$M' = (r_q p\Delta + r_q(1-p)\Delta) + 2r_{reg}\Delta,$$

since the cost of Large Queries reduces to 1 (sent to parent) and the registration cost increases to 2 (must register at 2 levels).

TE is beneficial, when $M' < M$ holds. Solving the inequality results in the condition $p > r_{reg}/r_q$, or $pr_q > r_{reg}$. This means that TE will reduce the overall number of messages coming to this subtree, when the rate of Large Queries exceeds the registration rate. Therefore, we set $T_{large} = r_{reg}/r_q$ in this setup. As an example, if query rate is 5 times of the registration rate, then $T_{large} = 20\%$.

By combining expansion and reduction actions, the band can move up and down the RST depending on the load. An additional advantage that band adaptation brings is that we no longer need a well-defined leaf level to support range queries. This is important because in many applications it may not be possible to predefine a fixed smallest granularity. For example, when the domain is real numbers, query ranges may be arbitrarily small. With band adaptation, the tree can grow downwards as deep as is needed to handle small range queries.

4.4. Protocol Overhead Analysis

The PMP plays a crucial role in supporting range queries efficiently. The overhead it introduces is fairly low. We now examine the overhead more carefully. From a node's point of view, in each round of PMP message exchange, it sends at most 1 PR and 2 PRR messages, and receives at most 2 PR and 1 PRR messages. Hence no node will be overwhelmed and no system bottleneck is created. From the system's point of view, the overhead of the protocol is determined by two factors: PMP message size and message exchange frequency.

The PMP messages are reasonably small. In the first round of message exchange, a PR message carries the LBM's size (2 integers) for each node in its subtree, so the message size is $O(n)$, where n is the number of leaves. As an example, for an RST with 200 leaves, the largest PR (the one to the root) has a size of ~ 1600 bytes. The PRR messages are even smaller with a size of $O(\log n)$, since it contains the path component. In the example, the largest PRR messages are the ones to the leaves; they have a size of ~ 70 bytes. The message size can be further reduced in future PMP rounds by only updating matrices that have changed.

The frequency of the periodic PMP message exchange can be set fairly low. When the load in the system is stable, the band does not change. The band may change when the matrices' size change due to significant load change, or when adaptation actions are needed due to changes in query ranges. Both of these occur on a much larger time scale in comparison to the registration or query rate.

The overhead of band adaptation is minimal since the decisions are made locally and no network wide message exchange is required. During band adaptation, an endpoint may get stale information about the tree. The impact of this transient state is small. For example, stale information may result in some additional, unnecessary registrations, or cause an endpoint to repeat its query if the node it originally contacted left the band.

5. Evaluation

In this section, we present evaluation results obtained from simulations. We implemented the range query mechanisms in an event-driven simulator developed in [9]. The simulator allows us to set up an overlay network with a configurable number of nodes, each of which can handle events related to registrations and queries. The simulator supports the load balancing mechanisms described in Section 2.2.

The simulator assumes the existence of an underlying DHT-based overlay for routing and forwarding. The cryptographic function SHA-1 is used as the system-wide hash function, $\mathcal{H}$. We do not explicitly consider the effect of node churn [5] in our evaluation for two reasons. First, recent work [19] shows that DHTs can work well under high churn rate. Second, the type of applications we are targeting are mostly infrastructure services, e.g., distributed monitoring, which typically have a fairly stable core overlay network.

5.1. Methodology

Our experiments use an overlay network with $10,000$ nodes. We assume a node's performance in handling registrations and queries is limited by its link bandwidth rather than its computation power. Each node is configured with a $500Kbps$ link (DSL level) for registrations and queries. Accordingly, each node sets thresholds of $50reg/sec$ and $200q/sec$ as the maximum sustainable registration and query rate, which correspond to 1000-byte and 250-byte registration and query message sizes respectively. When a node observes that one of these thresholds is crossed, its corresponding matrix will start to expand.

We use synthetic workloads to drive our simulations. Each registration load is comprised of a set of content names, each consisting of one AV-pair. The AV-pairs share the same attribute, a, which can take on 200 different values. This attribute's RST has 9 levels ($\lceil \log 200 \rceil + 1$). The sender of a name is selected randomly from the nodes in the system and the names' arrival times are modeled with a Poisson distribution. Query loads are similar, except that instead of one value, a range may be specified.

In addition to the **Static RST** and **Dynamic RST** designs as described in Section 3 and 4, the evaluation also considers the following algorithms:

- **Root Only**: All registrations and queries are sent to the root level nodes, i.e., there is no RST.
- **Leaf Only**: Registrations are sent to leaf nodes only, and queries are decomposed into point queries.
- **RST(3)**: Similar to Static RST, but a range with length R_q is decomposed into three adjacent ranges at level $\lfloor \log R_q \rfloor$ in the RST.

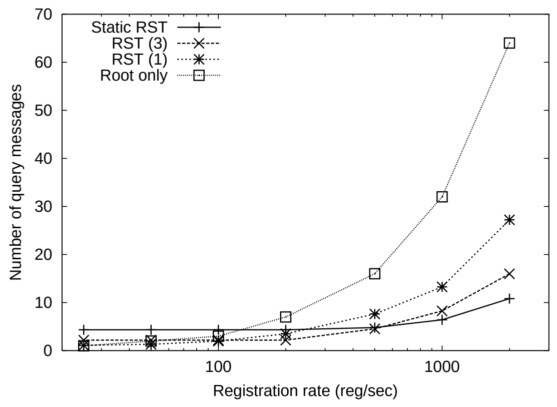


Figure 8. Query cost comparison.

- **RST(1)**: A query always uses the node in the RST that corresponds to the common prefix of its range's two end values. In this case, even a small range may correspond to a high level node, e.g., range $[3, 4]$'s prefix node is the root $[0, 7]$ in Figure 2.

We use the number of messages needed for registrations and queries as the primary metric to evaluate the system.

5.2. Performance of Static RST

We start by examining the performance of the static RST design. We first evaluate the query performance. In each experiment, we inject a registration load into the system with a certain rate, and then inject a query load with rate $200q/sec$. The registration rate varies from $20reg/sec$ to $2000reg/sec$. In the query load, there are 10,000 random queries but all have a range of $R_q = 20$. We compute the average number of query messages needed for each query after each run. We plot the results in Figure 8 as a function of the registration rate.

When the registration load is low, Static RST uses the most query messages due to its logarithmic decomposition. As the registration load increases, nodes higher in the tree will start to create partitions. As expected, the cost in Root Only grows linearly as partitions are created proportionally to the registration load. Since RST(1) also may use levels higher in the tree, its cost grows fast as well and it becomes more expensive than Static RST. Static RST also performs better than RST(3) under high load, since its decomposition is finer and uses more lower level nodes.

Next we examine the registration performance. In this set of experiments, we fix the registration rate and vary the query rate from $100q/sec$ to $5000q/sec$. Figure 9 shows the average number of registration messages needed for each registration as the query rate increases. For low query load, all the RST cases use more registration messages than Root Only, because they have to register with all 9 levels. As the

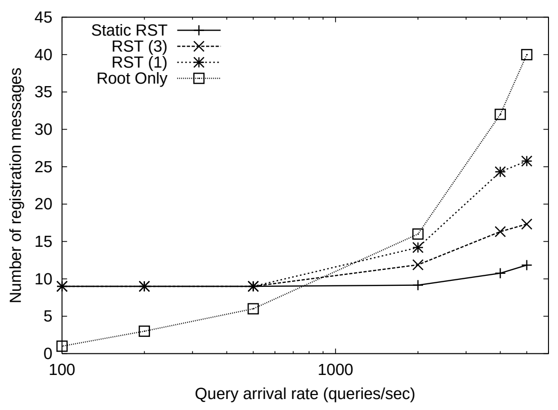


Figure 9. Registration cost comparison.

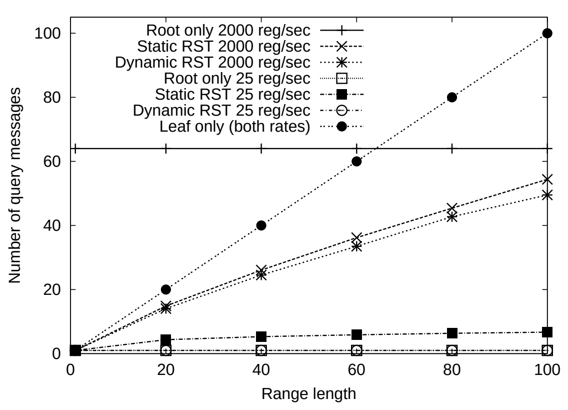


Figure 10. Query cost vs. Range length.

query load increases, the cost of Root Only grows linearly and it becomes the worst performer. Static RST performs the best. The reason is that for high query load, the registration cost of $\{a = v\}$ is dominated by the number of replicas along $Path(v)$. Recall that the *relevance* of an algorithm indicates how well a query matches the nodes that are being queried. For low relevance algorithms such as Root Only and RST(1), queries are concentrated at a small number of nodes with large ranges and cause them to replicate often. In comparison, in Static RST ($r = 1$), the query load is spread out, and the number of replications in the system is minimized.

In summary, the Static RST mechanism provides the best performance for both queries and registrations under high load. However, when the load is low, it performs poorly. This is consistent with our analysis in Section 3.4.

5.3. Performance of Dynamic RST

Next, we evaluate the performance of the Dynamic RST design. For the results we show here, we use the root level as the default band, and take measurements after necessary band adaptations complete. Due to limited space, we only show the query performance; the registration performance displays similar trends.

Figure 10 shows the average number of query messages as a function of the query range. In this set of experiments, we first inject a registration load into the system and then issue a query load with rate $200q/sec$. In each experiment, the range lengths are the same, and across experiments, the range varies from from 1 (point query) to 100 (50% of the domain). We use two registration loads with low ($25reg/sec$) and high arrival rates ($2000reg/sec$).

With low registration load, the root level (and all other levels) has only 1 partition. The Root Only and Dynamic RST designs perform the best, since they will just use the root for all queries. The Static RST design ignores the load status and always decomposes the query into logarithmic number of sub-queries, so the number of query messages grow logarithmically with the range length. The Leaf Only is the worst, and the cost grows linearly with the range.

Under high registration load, Root Only performs poorly, since irrespective of the range length, 64 query messages are needed for all queries (the root level has 64 partitions under this load). In the Leaf Only case, the number of query messages again grows linearly with the range length, since no partitions were created at the leaf level. The Static RST approach grows faster than logarithmic due to the partitions created at higher levels, but it still requires far fewer query messages than Root Only. The Dynamic RST design improves performance further since it does not need to decompose the query all the way to the leaf level. Of course, if the query range is the full domain, both RST approaches will degenerate into the Root Only design.

Using the same setup, Figure 11 compares the query performance as the registration load increases. We use two query loads, with a range length of 20 and 100 respectively. We plot in log-log scale to amplify the differences for low loads. Dynamic RST approach does the best in all cases. In particular, it tracks the Root Only case when the load is low by using high level nodes and avoiding unnecessary decomposition. It migrates towards and stays under the Static RST curve as the load increases.

5.4. System Optimization with Band Adaptation

We further evaluate how band adaptations improve the system's performance. In this experiment, initially the band only contains the lowest level (Level 0), and one can think of this as a result of previous point queries. We then inject query load with range 20 into the system. The arrival rate of queries in the query load varies from high ($2000q/sec$) to low ($20q/sec$) and finally to high ($2000q/sec$) again. The thresholds we used to trigger expansions are 20%.

Figure 12 shows the number of sub-queries received at each level for every 100 queries that are issued by end-

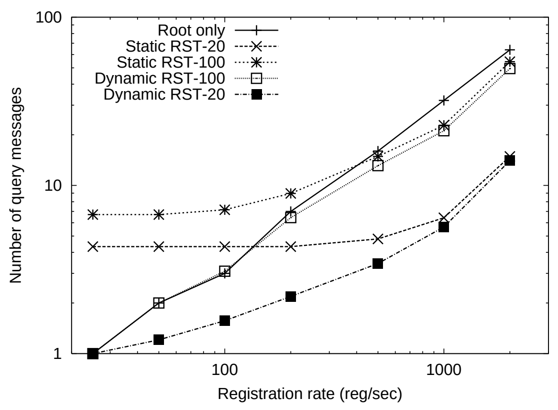


Figure 11. Query cost vs. Registration load.

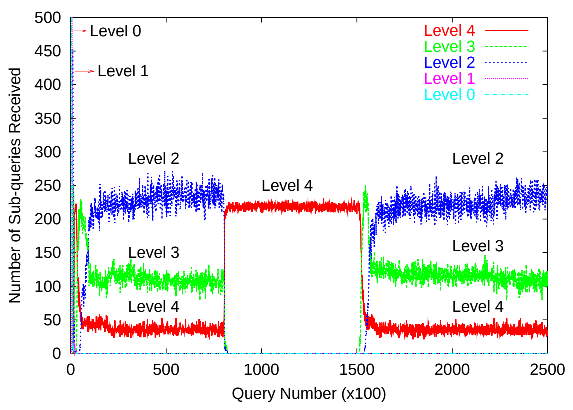


Figure 12. Band adaptation behavior.

points. The figure has three main sections. In the beginning, Level 0 receives all the queries, since it is the only level in the band, and each query is resolved using 20 sub-queries. As more queries arrive, since these queries are Large Queries with respect to Level 0, the band expands to higher levels, and eventually Level 4 nodes (with a length of 16) are added to the band. At the stable state, queries are only sent to Levels 2-4, and the average number of query messages needed drops to 3.8. With the high load ($2000q/sec$), LBMs in these levels all have multiple replicas. Next, the query rate drops to $20q/sec$, and LBMs shrink to only 1 replica. The query decomposition algorithm directs queries to Level 4, rather than further using Level 2 and 3. As shown in the figure, nodes in Levels 0-3 do not see enough queries, and they eventually drop out of the band. The band reduces to Level 4 only. At this time, the query cost drops to ~ 2 (using 2 Level 4 nodes for each query). Finally, the query rate increases again, and the band grows downwards through bottom expansion to recruit Levels 3 and 2 back. This experiment shows that the adaptation algorithms successfully adapt the band based on load changes to reduce query and registration cost.

6. Related Work

Efficiently supporting range queries in DHT-based systems was posed as an open question in [11, 13]. There have been some recent efforts in addressing this problem. In [17], the Prefix Hash Tree (PHT) is proposed to support range queries. While the PHT is conceptually similar to the RST, there are important differences between the two systems. Unlike our system, where we store contents and resolve queries using multiple levels depending on the load and query ranges, the PHT is a trie, and only leaf nodes store contents. Queries are first sent to the node corresponding to the common prefix of the range and traverse down to the leaves. In our system no tree traversal is needed, and our evaluation shows that the logarithmic query decomposition is more efficient than using the common prefix node.

In [3], the authors use space filling curves as hash functions in CAN-based DHT systems [16] to address range queries. In this work, a query is first sent to a node within the range; that node then locally broadcasts the query. Our system works with no assumption on the type of DHT used. Queries are sent to nodes that contain potential matches, and no broadcasting is involved. In [10], a mechanism based on locality sensitive hashing function is used for range queries in the context of relational databases. However, unlike our system, only approximate answers that are similar to users' range queries are returned.

SkipNet [12] proposed a lexicographic order preserving DHT, and thus allows data items with similar values to be placed on contiguous nodes. This facilitates range search, but the number of nodes must be visited is still linear to the query range due to lack of aggregation. P-Grid [2] is a DHT in which nodes are organized based on a virtual distributed search tree similar to our RST structure. The critical difference is that in P-Grid, the tree structure is used for DHT routing purpose to locate a node that holds a given key in the identifier space, and as such, like other DHTs, it does not directly support range queries.

In a different but similar context, Li *et al.* [14] proposed a distributed mechanism to partition a multi-dimensional space using a data structure similar to kd-trees to support range queries in sensor networks. Our system can be readily extended to high dimensions to support multi-dimensional range queries.

In traditional parallel databases, a large relation is often partitioned among multiple disks [7]. A partitioning technique that works well for both point and range queries is to partition the relation based on data values. A centralized partition vector must be consulted before a query may be issued. In our system, if we consider the collection of all content names as a large relation, the RST based mechanism mimics the value-based mechanism by partitioning the relation multiple times with different granularity. The advantage of our design is that queries can be resolved in a fully distributed fashion without using a centralized partition vector.

7. Conclusions

In this paper, we presented an adaptive protocol to address the range query problem in DHT-based systems. Our algorithms utilize Range Search Trees for content registration and query resolution. Registrations are aggregated at different levels of the RST to facilitate queries with different range lengths. Queries are decomposed to a small number of sub-queries for efficient resolution. The system operates in a fully distributed fashion without creating bottlenecks. The RST is adaptive: nodes are only instantiated if their presence in the RST can lower the overall registration and query cost. Our extensive simulation shows that the system can optimize itself to handle range queries efficiently based on the query ranges and load it observes while incurring low overhead.

Acknowledgement

We thank the anonymous reviewers and our shepard Dan Rubenstein for their helpful comments. We thank Shuheng Zhou for suggesting applying hash functions to ranges in a way similar to the Merkle tree [15]. We thank Christos Faloutsos, Leejay Wu and An-Cheng Huang for their valuable comments. This research was sponsored in part by the Defense Advanced Research Project Agency and monitored by AFRL/IFGA, Rome NY 13441-4505, under contract F30602-99-1-0518, and in part by the NSF under award number CCR-0205266. Additional support was provided by Intel. Jun Gao was also in part supported by a Siebel Scholar Award.

References

[1] Project IRIS, http://iris.lcs.mit.edu.

[2] K. Aberer. P-Grid: a Self-Organizing Access Structure for P2P Information Systems. In *Proceedings of the Sixth Intenational Conference on Cooperative Information Systems*, Trento, Italy, 2001.

[3] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *Proceedings of P2P 2002*.

[4] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of Pervasive 2002*, Zurich, Switzerland, August 2002.

[5] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *Proceedings of SIGCOMM 2003*.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of SOSP 2001*, Banff, Canada, October 2001.

[7] D. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[8] J. Gao and P. Steenkiste. Efficient Support for Range Queries in DHT-based Systems. Technical Report CMU-CS-03-215, Carnegie Mellon University, Dec. 2003.

[9] J. Gao and P. Steenkiste. Design and Evaluation of a Distributed Scalable Content Discovery System. *IEEE Journal on Selected Areas in Communications*, 22(1):54–66, January 2004.

[10] A. Gupta, D. Agrawal, and A. E. Abbadi. Approximate Range Selection Queries in Peer-to-Peer Systems. In *Proceedings of CIDR 2003*.

[11] 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 IPTPS'02*.

[12] N. Harvey, M. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of USITS'03*.

[13] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with pier. In *Proceedings of the 29th VLDB*, 2003.

[14] X. Li, Y. Kim, R. Govindan, and W. Hong. Multi-dimensional Range Queries in Sensor Networks. In *Proceedings of SenSys'03*.

[15] R. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Proceedings of Crypto'87*.

[16] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proceedings of SIGCOMM 2001*, pages 161–172, San Diego, CA, August 2001.

[17] S. Ratnasamy, J. Hellerstein, and S. Shenker. Range Queries over DHTs. Technical Report IRB-TR-03-009, Intel Corp., June 2003.

[18] P. Reynolds and A. Vahdat. Efficient Peer-to-Peer Keyword Searching. In *Proceedings of Middleware 2003*, Rio de Janeiro, Brazil, June 2003.

[19] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proceedings of USENIX 2004*.

[20] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proceedings of SOSP 2001*.

[21] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM 2001*, pages 149–160, San Diego, CA, August 2001.

[23] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-organizing Semantic Overlay Networks. In *Proceedings of SIGCOMM 2003*.