A Case Study in Building Layered DHT Applications

Paper #318

Abstract

Recent research has convincingly demonstrated that one can use Distributed Hash Tables (DHTs) to build scalable, robust and efficient applications. One question that is often left unanswered is that of simplicity of implementation and deployment. In this paper, we explore a case study of building an application where concerns of ease of deployment dominated the need for high performance. The application we focus on is Place Lab, an end-user positioning system. We evaluate whether it is feasible to use DHTs as an application-independent building block to implement a key component of Place Lab: its "mapping infrastructure." By strictly layering the application on top of a generic DHT service, we were able to decouple the deployment and management of Place Lab from that of the underlying DHT. In the paper, we demonstrate the flexibility of narrowly defined and application-independent DHT APIs to support this application. We identify the characteristics of the Place Lab application that made it amenable for deploying in this layered manner and comment on its effect on performance.

1 Introduction

Distributed Internet-scale applications are typically designed with the following characteristics in mind: scalability, availability, and robustness. An issue that is frequently overlooked is simplicity of implementation and deployment. Yet, in practice, this is often an equally important and difficult challenge. This is particularly true of recent peer-to-peer systems that are highly distributed in both location and administration.

This paper describes the design and evaluation of an application in which concerns over ease of operation dominated the need for high performance. The application, Place Lab, is an end-user positioning service for location-enhanced applications [19]. Place Lab clients estimate their physical location by listening for nearby radio beacons such as 802.11 access points and GSM cell towers in conjunction with a database of known beacon locations. The beacon database was initially designed as a centralized "mapping service." However, as the system gained popularity—since March 2004, users have downloaded the Place Lab software from over 3900 unique locations—concerns of privacy and ownership of the location database required transitioning to a decentralized architecture composed of mapping servers distributed across organizational domains.

What makes Place Lab's mapping service an interesting case study is that Place Lab's operators, a small group of UbiComp researchers, wished to limit the implementation and deployment overhead involved with providing a fully decentralized infrastructure. So we ask the question whether it is possible to isolate the Place Lab developers from the distributed application's deployment, management and robustness concerns.

A powerful design principle that is commonly used to simplify the construction of complex systems is that of *layering*. For example, the strict layering between IP and TCP allows the network to handle the complex operations of packet delivery and the end-hosts deal with reliability and congestion control. At a higher layer, Distributed Hash Tables (DHTs) are often cited as playing a somewhat similar role in building decentralized applications. Building an application on top of a DHT frees designers from having to address issues of scalability and robustness directly. Such an approach, if viable, would greatly simplify the building of distributed applications such as Place Lab.

There are many recent examples of DHT-based applications. In terms of whether they modify the routing algorithms underneath the DHT interface, these applications adhere to the principle of layering to varying degrees. Typically, applications like CFS [9], i3 [27] and PAST [10] which make fairly straightforward use of a DHT for simple rendezvous or storage are easy to implement in a layered fashion. On the other hand, systems such as Mercury [7] and CoralCDN [12] have more sophisticated requirements, which they achieve by altering the underlying DHT mechanisms.

Place Lab's mapping service is closer to this second category of applications in that it has more complex requirements than simply storage and rendezvous. Specifically, Place Lab's application interface is based on geographic *range* queries, not exact-match lookups. Place Lab applications often run on impoverished devices such as phones and PDAs with limited storage capacity, hence they need to be able to download relevant segments of the beacon database as needed. For example, when a user arrives in a new city, his or her device will query the mapping service for all beacon data within that region.

In spite of these requirements, if we could easily layer Place Lab over an existing DHT, that would go a long way toward simplifying implementation. However, this would not simplify operation of the service; the Place Lab operators would still have to deploy and manage a full-fledged DHT which is arguably not an easy task. Hence, we decided to push the notion of layering a step further and outsourced the operation of the DHT altogether to a third-party DHT *service*. In this, Place Lab is unique. All of the above application examples deploy their own DHT infrastructure and many violate layering by modifying the underlying DHT.

Building on top of a third-party DHT service restricts the interaction between the application and the DHT to a narrow and well-defined API. It is precisely these conflicting needs—building a complex data structure while having to live with a narrow DHT interface—that we believe makes Place Lab a good (admittedly harsh) stress test for the claim of DHTs as a composable building block. This is an important question because a lot of the value of DHTs will lie in the validation of their flexibility as a re-usable programming platform for large-scale distributed applications.

In this paper, we describe the design and implementation of Place Lab's mapping service over the OpenDHT [2] service. We use our experience to answer the following three questions:

- Is it feasible to use a simple DHT service as a building block for a larger more complex application
- Can this application leverage the purported simplicity and deployability advantages of DHTs
- What is the performance impact of using application-independent DHTs for this application

We recognize that a single case study is not sufficient to answer the more general question of just how broad a class of applications can be supported on top of a strictly layered DHT. Rather, our results provide an initial insight into the requirements that applications beyond simple rendezvous and storage can impose on DHT infrastructures. Moreover these requirements arise from an application that is being actively used by members of the research community and other early-adopters.

One of our challenges was to address Place Lab's need for range-based queries while maintaining the application-independence of the DHT, that is, *to support such queries without modifying the DHT*. Our solution is called Prefix Hash Trees (PHTs), a distributed trie-like data structure that can be built on top of a generic DHT. A simple PHT can perform single-dimensional range queries and an extension using linearization techniques [16] allows us to perform multi-dimensional queries (and specifically 2-D geographic range queries). One can view PHTs both as a solution to the specific problem of range queries and as a more general exercise in supporting sophisticated data structures on top of generic DHTs.

Our experience with building Place Lab has mixed results. On the one hand we found that the simple DHT interface goes a long ways in supporting Place Lab's non-traditional and varied use of the DHT. Building Place Lab over a DHT (even a third-party DHT like OpenDHT) was relatively easy (2100 lines of glue code) and our system effortlessly inherited the scalability, robustness and self-configuration properties of the DHT. This would seem to validate our hope for a DHT-based "narrow waist" for networked systems. However, a simple put-get interface was not quite enough. In particular, OpenDHT relies on timeouts to invalidate entries and has no support for atomicity primitives. Explicit invalidation of DHT entries and simple atomicity primitives are both application-independent and relatively straightforward extensions to the basic DHT API, so it should be possible for a third-party DHT implementation to support them. Thus, we remain hopeful that such sophisticated applications can be layered on top of a DHT service, but think that DHT services should slightly broaden their interface.

In return for ease of implementation and deployment, we sacrificed performance. With the OpenDHT implementation, a PHT query operation took a median of 2–4 seconds. This is due to the fact that layering entirely on top of a DHT service inherently implies that applications must perform a sequence of put-get operations to implement higher level semantics with limited opportunity for optimization within the DHT. Whether this loss of performance is a worthy tradeoff for ease of deployment is something that individual application developers will have to assess based on their specific requirements.

The rest of the paper is organized as follows. We discuss related work in Section 2. Section 3 describes Place Lab and its requirements from the DHT framework, while Section 4 presents details on the Prefix Hash Tree data structure. In Section 5 we discuss our experimental results, highlight the lessons learned in Section 6 and finally conclude in Section 7.

2 Related Work

There has been a variety of related work in DHT-based applications, in techniques for distributed range queries, and in the use of trie-based schemes in networking. Place Lab is by no means the first application to be built on DHTs. But we believe it is unique in using DHTs not only for traditional key-based lookup, but also as a building block for implementing a data structure with richer functionality (PHT) while still retaining the simple application-independent API of the DHT.

2.1 Other DHT-based Systems

An early and significant class of DHT-based applications are storage and rendezvous systems, including PAST [10], OceanStore [18], i3 [27]and those based on Chord's DHASH layer [28] (for example, CFS [9], Ivy [23] and Usenet-DHT [26]). Although these applications make straightforward use of the DHT, their implementations are not always decomposable from their underlying DHT. Scribe [25] and SDIMS [31] use the DHT topology to construct trees for multicast, anycast and aggregation. PIER [15] uses DHTs for relational database and file-sharing queries, extending the DHT beyond its basic put/get semantics to support query dissemination, as well as join and aggregation operations. Lastly, systems like CoralCDN [12] and POST [21] support large-scale applications by building custom DHTs underneath. What distinguishes Place Lab from all of these applications is its strict use of a layered approach by building entirely on top of the OpenDHT service.

2.2 Peer-to-Peer Range Queries

In recent years there has been a flurry of work on providing peer-to-peer range query functionality. We believe that the PHT scheme we describe here stands out because it is built without modifying the internal routing or topology of the DHT. This clean layering makes it easy to implement over third-party DHT infrastructures, and allows DHTs to support multiple functionalities, without being tuned specifically for range search.

In comparison, the Mercury system [7], Karger and Ruhl's item balancing [17], and Ganesan et al's online balancing work [13] explicitly load-balance the distribution of items by including specific modifications to the behavior of the underlying DHT. Typically, with evolving applications and data sets, this can induce churn in the DHT. PHTs on the other hand are built entirely on top of an existing DHT and rely on the spatial distribution of the data to achieve load balancing.

Aspnes and Shah [4] have proposed *skip graphs*, a distributed data structure that implements range search. However, maintaining load balance while mapping items to peers in the network requires non-trivial extensions to skip graphs [3]. In contrast, the PHT is based on a trie data structure, whose simplicity allows for a simple realization over a network of peers, as is demonstrated in this paper. Other related work includes a DHT-based caching scheme [14]; P-tree [8], a special-purpose P2P range-search structure; and a technique specifically for the CAN DHT based on space-filling curves [29].

2.3 Trie-based peer-to-peer systems

Cone [6] is a DHT-inspired, trie-based distributed data structure that is used to evaluate aggregation operators, such as MIN, MAX and SUM, over keys in a DHT. Cone is similar to PHTs in that it is based on a trie, but it is designed for aggregation operations, not range queries. In comparison, the PHT can not only perform range queries, but is easily capable of evaluating aggregation operators over elements satisfying a range predicate.

P-Grid is a DHT-like peer-to-peer lookup system that uses a trie-based approach at its core [1], along with a network of randomized links. It is quite different in design spirit from the PHT, which is a data structure layered on top of any DHT.

Finally, in independent work, Yalagandula [30] has proposed a trie-based scheme that is similar to our PHT proposal.[1] In this paper, we have explored beyond the basic concept of a PHT and built and deployed it for a real application.

3 Place Lab

3.1 Overview

Place Lab [19] is a radio-beacon based device positioning system that runs on commodity laptops, PDAs and cell phones. Client devices listen for broadcasts of *beacon* messages from nearby 802.11 access points and GSM cell towers. They estimate their own position by looking up these identifiers in a beacon database that maps identifiers to beacon location. Using locally cached segments of the database, clients can position themselves with a median accuracy of 12–40 meters depending on beacon density [19].

Inputs for the mapping database can come from many sources. Organizations like companies and universities often know the locations of their 802.11 access points. Another source of data is the *war-driving* community: driving around a neighborhood with a mobile computer equipped with a GPS device and a radio (typically an 802.11 card) in order to collect a trace of beacon availability. A simple strategy would be to use a centralized repository for all mapping data. In fact, such centralized databases already exist (*www.wigle.net*). However, as this infrastructure grows and becomes more popular, a single central authority will raise numerous concerns about privacy, ownership, and access. Since the mapping database is critical for clients to compute their own location, in some ways, centralizing it would be analogous to using a single centralized DNS server for clients to resolve DNS names.

Instead, the Place Lab researchers proposed a decentralized architecture for Place Lab's mapper service where any number of organizations, each with their own mapping servers, can host a portion of the mapping database. A simple way to achieve this would be to distribute the data geographically. But, this raises issues such as: how does a new mapping server pick the area for which it is responsible; who gets to be the mapping server for high-profile areas such as, say, Manhattan; how do war drivers determine which server is responsible for which region?

A different approach is to distribute the mapping data "randomly" across the set of mapping servers by making each server responsible for a random portion of the beacon identifier (MAC address or cell-id) key space. This method eliminates the need to assign servers by geography. Moreover, it ensures that even if data from a mapping server is lost or compromised the density of beacons managed by other servers will still allow clients to produce accurate location estimates.

This organization is well-suited for implementation on top of a DHT. Given Place Lab's requirements of ease of deployment, we built the service on top of OpenDHT [2], a third-party DHT service that provides a simple put/get interface to its applications. The Place Lab infrastructure is composed of a number of independent mapping servers interconnected through OpenDHT. Place Lab has three key requirements: routing data to and from the individual Place Lab servers, efficiently gathering data relevant to specific geographic regions, and most importantly simplifying deployment, robustness and availability. The rest of this section details how the DHT service supports these requirements.

3.2 Content-based Routing

For the most part, the processing of estimates for a single radio beacon is independent of estimates for other beacons. Accordingly, we distribute the mapping data in a manner such that data for a single beacon is always hosted by a deterministic mapping server and all war-driving readings for that beacon are forwarded to that mapping server.

DHTs provide a natural mechanism for achieving this distribution. We map beacon identifiers to SHA1 [11] keys. Each mapping server is responsible for a well-defined portion of the key space. To allow mapping servers to register

[1] After initially proposing the PHT idea, we came across this completely independent proposal [30], similar to the PHT. Our original proposal predates this work. However, neither is published as yet.

with the DHT and for clients to route war-driving records to appropriate mapping servers, we use the OpenDHT ReDiR mechanism [2]. ReDiR maintains a hierarchy of rendezvous points that allows clients to lookup the appropriate server(s) for their records. It can be implemented entirely by the mapping servers and their clients using the simpler put/get interface of the DHT. Effectively, the DHT provides the routing primitives that clients use to locate mapping servers.

3.3 Indexing for Retrieval

When a Place Lab client enters a new area, he or she must first download the mapping data for the new region. This involves performing a geographical range query over the mapping data. Such queries can be of the form "fetch all access points within 50 kilometers of my current position" or queries for well-known regions: "fetch all access points in the Seattle metropolitan area." Rather than allow arbitrarily complex query regions, we restrict queries to rectangular bounding boxes.

The underlying DHT's routing algorithm spreads beacon data uniformly across mapping servers with no semblance of spatial locality; yet, such locality is important to perform the above query efficiently. Prefix Hash Trees (PHTs) are our solution to efficiently coalesce estimated positions of nearby radio beacons. When a mapping server updates its estimate of a beacon's location based on new war driving readings, it also updates the PHT. For efficiency, these updates can be batched up and performed lazily. We will discuss this data structure and its implementation on top of OpenDHT in detail in Section 4.

3.4 Deployability, Robustness and Availability

OpenDHT provides the routing, storage, and robustness substrate for Place Lab. Individual mapping servers (from possibly independent organizations) connect directly to the DHT. They rely on the DHT to provide much of the robustness and availability. The servers store the current estimates of each radio beacon's location in the DHT. The DHT handles replication and recovery, and for the most part, Place Lab does not need to deal with any of these issues. In particular, if a mapping server fails, the DHT routing mechanisms automatically ensure that the failed server's successor in the routing overlay takes over responsibility for the failed server's key space. The mapping server's administrator still must handle restarting of the failed server, but the DHT provides automatic graceful fail-over in the meanwhile.

Mapping servers periodically refresh their data in the DHT. This ensures that even in the event of catastrophic failure of the DHT where all replicas of a beacon's data are lost, the mapping servers will eventually recover them. Moreover, the temporary loss does not in practice affect the application performance. This resilience is due to both the temporal and spatial redundancy in the data. The effect of a lost information for a beacon is reduced by the likelihood that a new war driver may submit fresh information for the beacon eventually. Spatially, the impact of lost beacons is reduced by readings for other nearby beacons that map to different servers. As we will show in section 5.6, a loss of even 30% of the beacon data results in no noticeable reduction in positioning accuracy.

4 Prefix Hash Trees

We now look at the PHT data structure in detail. Unlike the various recent proposals for incorporating range query support into DHTs [7, 13, 17], Prefix Hash Trees are built entirely on top of the put/get interface, and thus run over any DHT, and specifically on a third-party DHT service like OpenDHT. Range queries use only the get(*key*) operation and do not assume knowledge of nor require changes to the DHT topology or routing behavior.

In its simplest form, a PHT is a trie-based distributed data structure that supports one-dimensional range queries. As we will show below, this can be extended to support the two-dimensional geographic range queries required by Place Lab. As a corollary, PHTs can also support heap queries ("what is the maximum/minimum?") and proximity queries ("what is the nearest element to *X*?"). PHTs are efficient, in that updates are doubly logarithmic in the size of the domain being indexed. Moreover, PHTs are self-organizing and load-balanced. They tolerate failures well; while they cannot by themselves protect against data loss when nodes go down, the failure of any given node in the Prefix Hash Tree does not affect the availability of data stored at other nodes. Moreover, PHTs can take advantage of any replication or other data-preserving technique employed by a DHT.

4.1 The Data Structure

A Prefix Hash Tree assumes that keys in the data domain it is indexing can be expressed as binary strings of length *D*. It is fairly straightforward to extend this to other alphabets through multiway indexing, or by encoding them in binary. A PHT is essentially a binary trie in which every node corresponds to a distinct prefix of the data domain being indexed. Each node of the trie is labeled with a prefix that is defined recursively: given a node with label *L*, its left and right child nodes are labeled *L0* and *L1* respectively. The root is labeled with the attribute being indexed, and downstream nodes are labeled as described above.

Each node in the PHT has either zero or two children. Keys are stored only at leaf nodes. Unlike a binary search tree, all keys that are stored in the same leaf node share the leaf node's label as a common prefix. The PHT imposes a limit *B* on the number of keys that a single leaf node can store. When a leaf node fills to capacity, it must *split* into two descendants. Similarly, if keys are deleted from the PHT, two sibling leaf nodes may *merge* into a single parent node. As a result, the shape of the PHT depends on the distribution of keys; it is "deep" in regions of the domain that are densely populated, and conversely, "shallow" in regions of the domain that are sparsely populated.

As described this far, the PHT structure is a fairly routine binary trie. What makes the PHT interesting lies in how this logical trie is *distributed* among the servers that form the underlying DHT. This is achieved by hashing the prefix la-

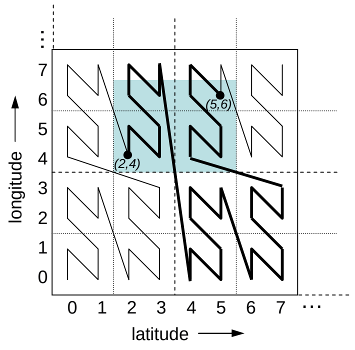
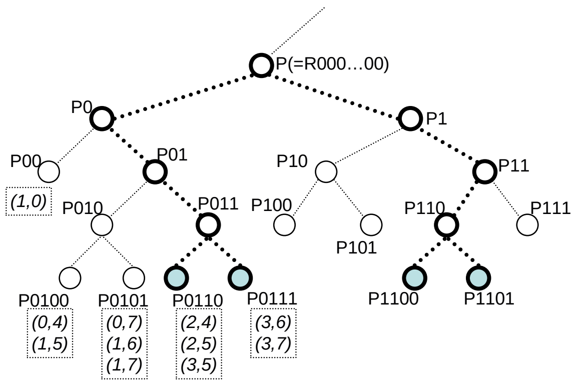


Figure 1: *Recursive shape of a z-curve linearization to map a two-dimensional coordinate space into a one-dimensional sequence. The shaded region represents a two-dimensional range query for data points in the space (2,4)–(5,6). The bold line represents the corresponding one-dimensional range in the z-curve between the lowest and highest linearization points of the original query.*



PHT node label	(Lat,lon)	(Lat,lon) *binary*	Z-curve key
P00	(1,0)	(001,000)	000010
P0100	(0,4) (1,5)	(000,100) (001,101)	010000 010011
P0101	(0,7) (1,6) (1,7)	(000,111) (001,110) (001,111)	010101 010110 010111

Figure 2: *A portion of a sample two-dimensional PHT for Place Lab. The table shows the data items (and their z-curve keys) that are stored at some of the leaf nodes in the PHT. As shown, each data item is stored at the unique leaf node whose label is a prefix of the item's z-curve key.*

bels of PHT nodes over the DHT identifier space. A node with label L is thus assigned to the DHT server to which L is mapped by the DHT hashing algorithm, that is, the server whose identifier is closest to $hash(L)$. This hash-based assignment implies that given a PHT node with label L, it is possible to locate it in the DHT via a single get(). This "direct access" property is unlike the successive link traversal associated with typical tree-based data structures and results in the PHT having several desirable features that are discussed later in Section 4.7.

4.2 Adapting PHTs for Place Lab

Queries in Place Lab are performed over the two-dimensional latitude-longitude coordinate domain ($-180.0 < longitude < 180.0$, $-90.0 < latitude < 90.0$). To index this domain using PHTs, we rely on a technique that maps multi-dimensional data into a single dimension. This approach is known as *linearization* or *space-filling curves*, and well-known examples include the Hilbert, Gray code, and Z-order curves [16]. First, we normalize all latitudes and longitudes into unsigned 40-bit integer values, which in turn, can be represented using a simple binary format. We then use the z-curve linearization technique to map each two-dimensional data point into an 80-bit one-dimensional key space. We chose z-curves because they are simple to understand and easy to implement. In Section 5.4, we will compare the performance of the various linearization techniques. Z-curve linearization is performed by interleaving the bits of the binary representation of the latitude and longitude. For example, the normalized point (2,3) would be represented on the z-curve using the 80-bit key 000...001101. Figure 1 shows the zig-zag shape that such a z-curve mapping takes across the two-dimensional coordinate space.

The PHT for two-dimensional queries uses these z-curve keys and their prefixes as node labels. Due to the interleaving of the latitude and longitude bits in a z-curve key, each successive level in the PHT represents a splitting of the geographic space into two, alternately along the latitude axis and then along the longitude axis. Data items (tuples of the form $\{latitude, longitude, beacon\text{-}id\}$) are inserted into the leaf node whose label is a prefix of the z-curve key associated with that latitude-longitude coordinate. Figure 2 shows a sample PHT along with an example assignment of data items to PHT leaf nodes assuming three-bit normalized latitude and longitude values.

4.3 PHT Operations

Now that we have described what the PHT data structure looks like, let us focus on the various operations needed to build and query this data structure using a DHT.

4.3.1 Lookup

Lookup is the basic primitive used to implement the other PHT operations. Given a key K, a PHT lookup operation returns the unique leaf node $leaf(K)$ whose label is a prefix of K. A lookup can be implemented efficiently by performing a binary search over the $D+1$ possible prefixes corresponding to a D-bit key. Starting at the prefix of length $D/2$, each DHT get() returns one of three states: no PHT node found, interior node found, or leaf node found. If the current prefix is an internal node of the PHT, the search tries longer prefixes. Alternatively, if the current prefix is not a node of the PHT, the search tries shorter prefixes. The search terminates when the required leaf node is reached. An important feature of this lookup is that unlike traditional tree lookups, it does not require each lookup to originate at the root, thereby reducing the load on the root (as well as nodes close to the root). Minor modifications to this algorithm can be used to perform a lookup of a prefix P instead of a full-length key K.

Binary search requires $\lfloor \log{(D+1)} \rfloor + 1 \approx \log D$ DHT gets, which in turn is doubly logarithmic in the size of the data domain being indexed. This ensures that the lookup operation is extremely efficient. However, binary search has the drawback that it can fail as a result of the failure of an internal PHT node. The search may not be able to distinguish between the failure of an internal node, in which case search should proceed downwards, and the absence of a PHT node, in which case the search should proceed upwards. In such a situation, the PHT client can either restart the binary search in the hope that a refresh operation has repaired the data structure (see section 4.4), or perform parallel gets of all prefixes of the key K. The parallel search is guaranteed to succeed as long as the leaf node is alive and the DHT is able to route to it, and thus provides a fail-over mechanism. This suggests two alternate modes of operation, namely, low-overhead lookups using binary search, and low-latency fail-over lookups using parallel search.

4.3.2 Range Query

For a one-dimensional PHT, given two keys L and H ($L \leq H$), a range query returns all keys K contained in the PHT satisfying $L \leq K \leq H$. Such a range query can be performed by locating the PHT node corresponding to the longest common prefix of L and H and then performing a parallel traversal of its subtree to retrieve all the desired items.

Multi-dimensional range queries such as those required for Place Lab are only slightly more complicated. A query for all matching data within a rectangular region defined by (*latMin*, *lonMin*) and (*latMax*, *lonMax*) is performed as follows. We determine the linearized prefix that minimally encompasses the entire query region. This is done by computing the z-curve keys *zMin* and *zMax* for the two end-points of the query, and the longest common prefix *zPrefix* of *zMin* and *zMax*. We then look up the PHT node corresponding to *zPrefix* and perform a parallel traversal of its sub-tree.

Unlike the simpler case of one-dimensional queries, not all nodes between the leaf for the minimum key and the leaf for the maximum key contribute to the query result. This is illustrated in Figures 1 and 2 which show a query for the rectangular region (2,4)–(5,6). As shown in Figure 1, the linearized range between these two points (shown by the bold line) passes through points (and correspondingly PHT nodes) that are not within the rectangular region of the search. This is also depicted in the PHT representation in Figure 2: the leaves for the end-points of the query are *P0110* and *P1101* . Not all of the leaf nodes between these two can contribute to the query result. In fact, the entire subtree rooted at *P10* does not contain any data items that fall within the query range.[2]

Hence, the query algorithm works as follows: Starting at the PHT node corresponding to the longest common prefix *zPrefix*, we determine whether this node is a leaf node. If so, we apply the range query to all items within the node and report the result. If the node is an interior node, we evaluate whether its left subtree (with a prefix of *zPrefix*+"0") can contribute any results to the query. This is done by determining whether there is any overlap in the rectangular region defined by the subtree's prefix and the range of the original query. This check can be performed with no additional gets, so incurs almost no penalty if it fails. If an overlap exists, the query is propagated recursively down the left subtree. In parallel, we perform a similar test for the right subtree (with a prefix of *zPrefix*+"1") and if the test succeeds, propagate the query down that sub-tree as well. Thus the query algorithm requires no more than d sequential steps, where d is the depth of the tree.

Various heuristics can be applied to optimize the performance of PHTs. For example, for certain queries (such as a small range containing the midpoint of the key-space), it may be desirable to break the search query into two, and treat these sub-queries independently. In this example, the longest prefix match will be the root of the PHT. To avoid having to start traversing at the root, we observe that such a range can be sub-divided into two ranges, one in each sub-tree of the root. By handling these separately, it is possible to ensure a search starts at a level in the PHT that is appropriate for the query, that is, smaller queries start lower down in the PHT.

4.3.3 Insert/Delete

Insertion and deletion of a key K both require a PHT lookup operation to first locate the leaf node *leaf*(K). During insertion, if the leaf node is already full to its limit of B values, it must first be split into two children. In most cases, the $(B+1)$ keys are distributed among the two children such that each of them stores at most B. However it is possible that all $(B+1)$ keys will be distributed to the same child, thus necessitating a further split. To avoid this, the split operation determines the longest common prefix of all of the $(B+1)$ keys and creates two new leaf nodes one level deeper than that common prefix, thereby ensuring that each of the new leaves has no more than B keys. The keys are distributed across these two new leaves and all nodes in between the original node being split and the new leaves are marked as interior nodes. All of these operations can be parallelized for efficiency.

Similarly, when a key is deleted from the PHT, it may make it possible to coalesce two sibling leaf nodes into a single parent node. Although deletion can be implemented entirely by putting an invalidation marker for that key and waiting for OpenDHT to timeout the original key, it is inefficient since before the key times out, gets will still return the deleted item and its invalidation marker. A better approach is to add an explicit remove() operation to the underlying DHT. Note that remove() is not an operation specific to PHTs or Place Lab, and would potentially benefit other applications built on top of the DHT as well.

[2] *P10* contains items whose latitude coordinates are of the form 000...1XX and longitudes are of the form 000...0XX, that is, items in the range (4,0)–(7,3). This range does not overlap the query range and hence the entire subtree can be discounted.

4.4 Refreshing and recovering from failure

PHTs inherit all of the resilience and failure recovery properties of the underlying DHT. However, in the event of catastrophic failure of all replicas in the DHT, the PHT can lose data. Although the PHT algorithms are fairly resilient even in the face of loss of interior PHT nodes, one must eventually restore the lost data. To achieve this, we rely on soft state updates. Each PHT entry (leaf node keys and interior node markers) has associated with it a time-to-live (TTL). When the TTL expires, the entry is automatically deleted from the DHT.

Place Labs mapping servers periodically refresh the values that they have inserted into the PHT. All keys are inserted into the DHT with a TTL of T seconds. Every $T/2$ seconds, a mapping server refreshes its keys by resetting their TTL to T. At the same time, it checks the parent of the leaf node. If the parent's TTL has dropped to less than $T/2$ seconds, it refreshes the parent as well. This continues recursively until it reaches the root or a parent whose TTL is greater than $T/2$. Thus, interior nodes are refreshed only as needed. If an interior node is lost due to failure, it will eventually be refreshed as a consequence of the refresh of a value in one of its descendant leaf nodes.

4.5 Dealing with concurrency

The PHT as described above has potential race conditions that can result in (temporary) loss of data as well as duplication of work. For example, if two mapping servers attempt to insert keys K_1 and K_2 into the same leaf node, and that leaf node is full, both servers will attempt to split the leaf node resulting in duplicate work. A worse race condition can cause one server's insert operation to get lost while a different server has begun the process of splitting a leaf node. This however is a temporary problem since the refresh mechanisms described in the previous section will eventually recover the lost data.

These inefficiencies occur because the PHT is implemented entirely outside the DHT by the independent mapping servers. In the absence of concurrency primitives in the DHT, they cannot be eliminated. Hence, we added a localized atomic test-and-set mechanism to the OpenDHT API. Like the remove() extension, this extension is not PHT-specific and can potentially benefit many distributed applications. The test-and-set works as follows: get(*key*) returns a generation number for the key. This generation number is updated whenever the DHT key is modified. We use the modification timestamp as the generation number. In addition, we implemented a put_conditional(*key*, *value*, *gen*); the put succeeds only if the key has not been modified since the generation number *gen*.

To implement this concurrency primitive correctly in the presence of replication and failures, the DHT must provide strong guarantees for atomic writes. Etna [22] is a consensus protocol based on Paxos [20] that can provide such guarantees. However, the protocol is fairly involved and will significantly complicate the DHT implementation. Instead, our extension uses a much simpler mechanism that works in practice for the common case: serialize all put operations through the master replica for each key. In the event of churn, if multiple DHT nodes think they are the master replica for a key, this mechanism will fail. Such events will hopefully be rare for a DHT service and as mentioned earlier will only result in inefficiency in the PHT, not loss of correctness.

With these primitives, the insert operation is modified as follows. When inserting key K into *leaf*(K), we use the put_conditional() primitive to ensure that the leaf has not been modified or split since we performed the lookup. When a leaf node needs to be split, we first mark it as being *in transition* using the put_conditional primitive. If multiple servers attempt to split the same node, only one of them will succeed. All of the other PHT nodes that are involved in this split operation are then marked *in transition*. Only then is the split operation performed. The *in transition* markers are removed after the split operation has completed.

4.6 Caching to improve performance

Since the lookup primitive is central to all PHT operations, it can be optimized by using a client-side hint cache that keeps track of the shape of the PHT based on previous lookup operations. When a lookup for key K returns the leaf node L (a prefix of K), the cache records L as a leaf node and all entries from the root to the parent of L as interior nodes. A new lookup for a different key K' is first checked against this cached information. If the cache returns a leaf node L', the client performs get(L') to verify that the PHT has not been reconfigured and that the node is indeed still a leaf node. A cache hit thus generates a single DHT operation. Upon a cache miss, however, the lookup must revert to the binary search algorithm.

4.7 PHTs versus linked data structures

This section compares the merits of the PHT with balanced-tree-based indexes, such as the B-tree, with particular emphasis on implementation in a distributed setting. While tree-based indexes may be better in traditional indexing applications like databases, we argue the reverse is true for implementation over a DHT.

The primary difference between the a trie and a tree, is as follows: a trie partitions the *space* while a tree partitions the *data set*. The implication for PHTs is that it is possible to exploit the DHT to address a PHT node directly via a single get(). On the other hand, a tree-based data structure must perform a top-down traversal from the root to locate any key. This translates into several key advantages in favor of the PHT when compared to a balanced tree index.

Efficiency: A balanced tree has a height of $\log N$, where N is the number of elements in the tree; so a key lookup requires $\log N$ DHT lookups. For PHTs, the binary search lookup algorithm requires only $\log D$ DHT operations, D being the number of bits in the PHT key.

Load Balancing: As mentioned before, every lookup in a tree-based index must goes through the root, creating a potential bottleneck. In the case of PHTs, binary search allows the load to be spread over $2^{\frac{D}{2}}$ nodes (in the case of uniform

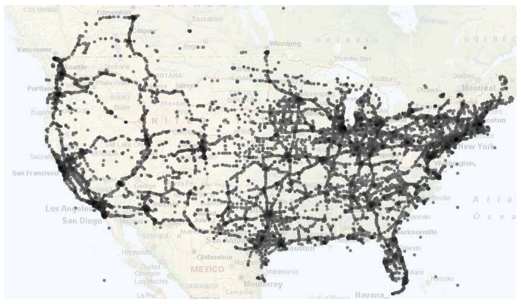


Figure 3: *Distribution of the input data set. The intensity of the dots on the map corresponds to the density of data points in that region.*

lookups), thus eliminating any bottleneck.

Fault Resilience: In a typical tree-based structure, the loss of an internal node results in the loss of the entire subtree rooted at the failed node. PHT however does not require top-down traversals; instead one can directly "jump" to any node in the PHT. Thus the failure of any given node in the PHT does not affect the availability of data stored at other nodes.

5 Evaluation

We now measure the performance of the DHT-based implementation of Place Lab's mapping service. The two main operations that Place Lab performs are: routing of beacon records from war drivers to mapping servers for updating beacon position estimates, and routing of beacon position estimates both to and from the PHT. The former is a fairly vanilla use of a DHT. Records are hashed based on each beacon identifier and this hash is used to redirect through the DHT to a mapping server. Accordingly, we focus our measurement effort on the prefix hash tree mechanism and the way it behaves both under insert loads from the mappings servers and under query loads from downloading clients.

5.1 Setup

We implemented PHTs and the rest of the Place Lab infrastructure on top of OpenDHT. The implementation effort required to build the glue between Place Lab's application code and the underlying DHT and to build a robust PHT implementation was surprisingly minimal. The code consists of 2100 lines of Java. Using this implementation, we conducted a series of experiments with the following setup: a deployment of OpenDHT with 24–30 nodes spread across machines on the US West Coast, US East Coast and England. Although our goal was to run Place Lab on an existing DHT service, we could not use the pre-deployed OpenDHT service for experimentation since we needed to use enhanced APIs (e.g., put_conditional()) and experiment with killing and restarting service nodes. In addition to our deployment, we repeated our experiments on a larger deployment using PlanetLab [24]. However, due to the vagaries of load on PlanetLab, the results from those experiments were erratic and are left out in this discussion.

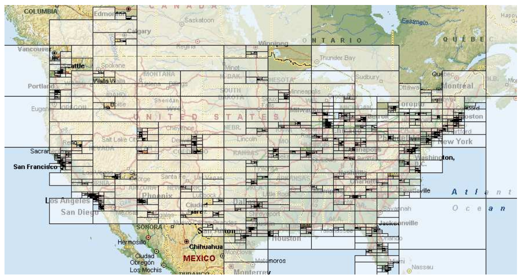


Figure 4: *The structure of the PHT for the 1.4 million input data set and a block size of 1000.*

As input, we used a data set composed of known locations of 1.4 million 802.11 access points gathered from a war-driving community web service, Wigle (*www.wigle.net*). This data set consists of estimated AP positions based on war drives submitted by users in the United States to the Wigle service. Figure 3 shows the distribution of the input data. We conducted experiments with different data set sizes picked uniformly at random from this larger set.

We constructed a query workload composed of 1000 queries to represent a set of typical Place Lab queries; the workload was proportional to the distribution of access points in the input data. We made this choice under the assumption that high access point density corresponds to higher population density and thus there is a higher likelihood of queries in those regions. Each query was generated by picking an access point at random from the input data set and building a rectangular region around the location of that access point with a size that was picked uniformly at random from [0–1.0] latitude/longitude units. Such a query corresponds to requests of the form: "I can hear access point X, find all APs within distance Y of this AP."

5.2 Structural Properties

In the first set of experiments, we constructed PHTs with progressively larger data sets and measured the structure of the resulting trees. Figure 4 shows a depiction of the PHT for the entire data set with a block size of 1000 overlaid on top of a map of the US. Each rectangle on the map represents a leaf node in the PHT. Comparing to the input data set shown in Figure 3, we note that areas with high AP density get sub-divided into smaller rectangular blocks than sparse areas. This is because we use a constant block size across the PHT. This organization ensures that queries for dense areas can be spread across a larger number of DHT nodes thereby reducing the bottleneck that popular queries may cause.

We measured the tree characteristics using two metrics: depth of the tree and block utilization (number of elements per PHT leaf node as a percentage of the block size).

Tree Depth: Figure 5 shows a CDF of the depth of leaf nodes in a PHT with 1.4 million elements and a block size of 1000. Between the 20th and 80th percentiles, the tree depth varies between 18 and 26. Some nodes in the densest part of the data set have higher depth (as deep as 33) while a small fraction of nodes in the sparse parts of the country are shal-

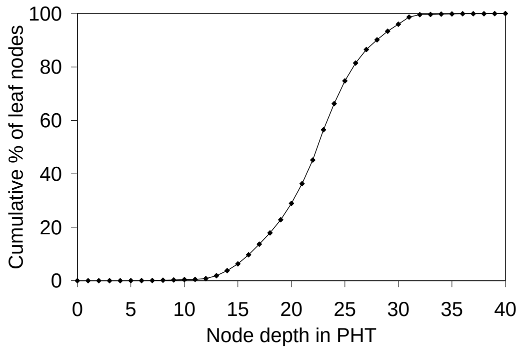


Figure 5: *A cumulative distribution function (CDF) of leaf node depth for a PHT with an input data set of 1.4 million and a block size of 1000.*

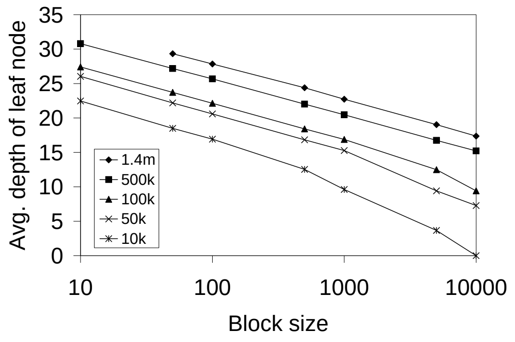


Figure 6: *Variation in tree depth as a function of block size for different input data sizes*

lower. Figure 6 shows the variation in average depth of the PHT for varying block sizes and different input data set sizes. From the figure, we can see that the tree depth decreases logarithmically with the block size, that is, larger block sizes result in shallower trees. With larger blocks, fewer accesses are needed to retrieve a portion of the data space, however, there is greater contention for nodes within the PHT. The figure also shows that (as one would expect) the tree depth increases with increasing data set sizes. Although not obvious from the figure, this increase is logarithmic as well.

Block Utilization: This experiment looks at how full the leaf nodes are as a percentage of the block size. Figure 7 shows the utilization as a function of block size for varying input data sizes. The plots for input data size of 50k and 100k show that the block utilization is high for small block sizes. It drops as the block size is increased, and eventually begins to grow again once the block size begins to approach the total input data size. The non-uniformity of the input data results in a skewed distribution of data across leaf nodes, and causes the average leaf utilization to be lower than if the data were uniformly distributed. Even with non-uniform data, at small block sizes, most blocks fill to capacity and thus the utilization in those cases is high. At very large block sizes (comparable to the input data set size), the tree becomes shallow and the non-uniformity of the data is averaged out, thus resulting in better block utilization.

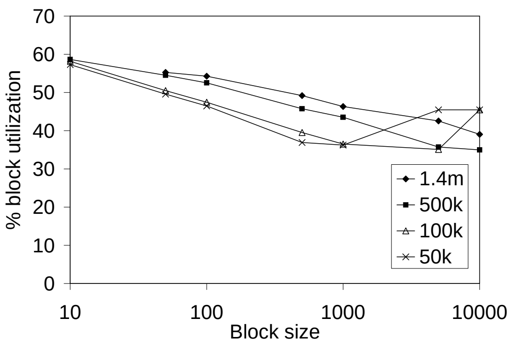


Figure 7: *Block utilization (number of items in a leaf node as a percentage of block size) versus block size for varying input data set sizes.*

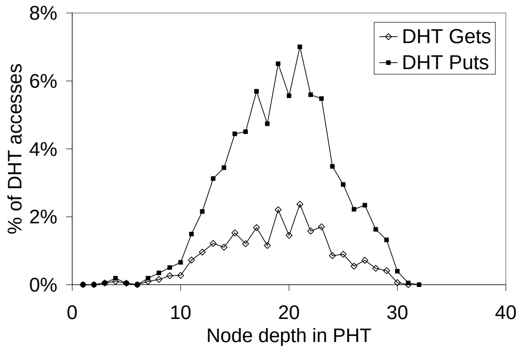


Figure 8: *A plot of the accesses (DHT*puts *and* gets*) to a PHT at each tree level while inserting items into the PHT. There are a total of 500,000 items in the PHT and the block size is 1000.*

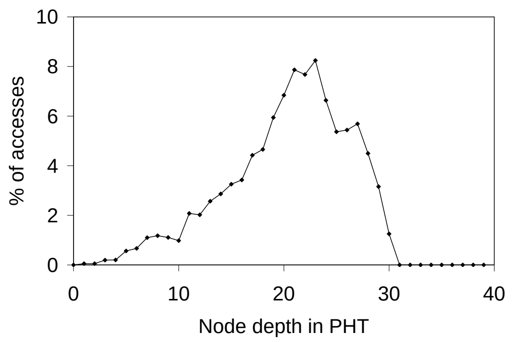


Figure 9: *A plot of the accesses (DHT* gets*) to a PHT at each tree level for a query workload. The PHT contains 500,000 access points and used a block size of 1000.*

5.3 Performance of the PHT

One critical advantage offered by a PHT over simpler data structures like a traditional pointer-based binary tree is that because of its structured key-space-based layout, PHT lookups can bypass the root and begin looking for data at lower levels in the tree. This offers PHTs the potential to avoid having the upper levels of the tree be hotspots that limit throughput. Figures 8 and 9 show the spread of DHT accesses across PHT levels for PHT insert (for 500,000 items)

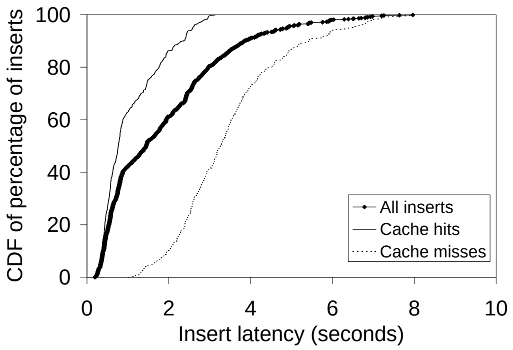


Figure 10: *A cumulative distribution function (CDF) plot of the percentage of insert operations as a function of the insert latency for a PHT with 100,000 items and block size of 500.*

Data size	Query time (sec)	Block size	Query time (sec)
5K	2.125	10	6.048
10K	2.761	50	4.524
50K	3.183	500	2.521
100K	3.748	1000	3.748

Table 1: *Variation in average query processing time for different input data set sizes (for a block size of 1000) and for varying block sizes (for an input data set size of 100K).*

and query operations (for the entire query workload) respectively. These graphs show that the levels of the tree close to the root are accessed very seldom, with the bulk of the activity in the depth range of 16 to 30. For sparse regions, and for queries for large-sized areas, the query starts higher up in the tree. Yet, the dominant accesses are for leaf nodes deep within the tree.

The previous charts only show the distribution of DHT operations across PHT nodes. The critical test of the viability of PHTs is the actual latencies required to perform insert and query operations. The next set of experiments evaluate this performance.

Insert Operations: For this experiment, we pre-loaded a PHT with 100,000 elements. We then started an insert workload composed of 1000 new randomly chosen elements and measured the performance of the insert operations. Figure 10 shows a CDF of the insert operations as a function of insert latency for a PHT block size of 500.

The graph also shows the effect of the lookup cache (Section 4.6). After the PHT had been pre-loaded, we started the client with an empty cache. Gradually as the client inserted more and more elements into the PHT, it discovered the shape of the tree and was able to skip the binary lookups by hitting directly in the cache.

We notice that the median insert operation takes about 1.45 seconds. When there is a cache miss, inserts take a median of 3.26 seconds, whereas on a cache hit, the median is 765ms. Part of the performance deficiency is due to a lack of optimization in the OpenDHT implementation. During a DHT get() operation, if a key matches a number of values, the current OpenDHT implementation returns only the first 1kBytes of those values and requires clients to perform additional get() operations to retrieve the remaining values. Hence, fetches of large leaf nodes can result in a cascade of a number of DHT-level operations. We have communicated this issue to the OpenDHT developers and a future version is expected to fix this by allowing bulk gets. With this and other minor optimizations of the DHT implementation, we expect the median insertion latency to be reduced by a factor of two.

Still, the insertion latency is not negligible. To a large extent, this is a result of our decision to build our range query data structure entirely on top of a general-purpose DHT service. A typical insert operation is composed of a binary search (median of 6 DHT gets in this experiment) followed by a put(). Some small number of insertions result in splits and thus have a higher latency. All of these operations are invoked from outside the DHT service, and hence cannot take advantage of any specialized routing within the DHT for efficiency.[3]

As we can see from the figure, techniques such as aggressive caching can help reduce latency substantially. In practice, for a workload like Place Lab's, we anticipate the PHT structure to stay mostly static and result in modifications (and consequently, potential cache invalidations) only when a new war drive is submitted into the system. Even then, it is typically expected that most war drives are local to a neighborhood and hence affect only one portion of the PHT. Thus with typical Place Lab usage, we expect the lookup cache to provide significant improvement for insert latencies.

Query Performance: Next, we look at the performance of the query workload. We pre-loaded the PHT with input data sets of varying sizes (as well as varying PHT block sizes). Table 1 shows the average query latencies as functions of the input data set sizes and block sizes. As one would expect, with larger data sets, queries take longer to resolve. However, even a jump of a factor of 20 (from 5K to 100K) in the data set size causes the query latency to increase only by a factor of 1.76. This is due to the parallelism and the logarithmic performance afforded by the PHT.

When we vary the block size, query latencies initially drop, since larger blocks implies fewer operations and fewer PHT nodes that need to be touched. However, if we keep increasing the block size, query latency starts to go up again. This is because at large block sizes, get() operations on PHT leaf nodes potentially return more items than the query actually matches. Note that this is a direct result of our decision to implement PHTs entirely using a third party DHT service. If we were to run PHT-specific code directly within the DHT nodes, we could have reduced this overhead by filtering values based on the query before returning them from the DHT.

Figure 11 shows a scatter plot of the query times for each of the queries as a function of the query response size in one run of our experiments (for an input data set of 100,000 elements and block size of 500). The graph shows the total time

[3]A direct comparison of PHTs to the performance of a customized system such as Mercury [7] would be ideal; unfortunately, the Mercury implementation is not yet available for distribution.

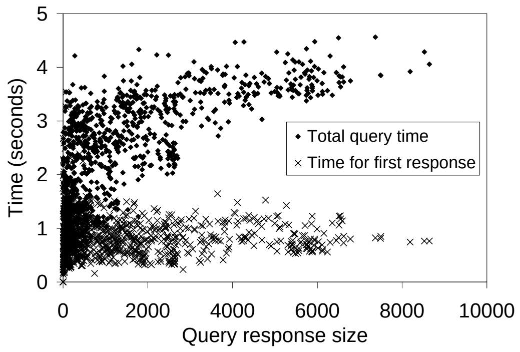


Figure 11: *A plot of the query response time (total time and time for first data item) as a function of response set size for a PHT with 100,000 items and block size of 500.*

for the query as well as the time taken for the first item of the results to reach the client. In general, queries with larger responses take longer. But even those queries return their first result within a second or so of issuing the query. For this experiment, the median query response time was 2.52 seconds while the median time for the first set of responses 0.8 seconds.

For our query experiments, the lookup cache did not provide as much benefit as for inserts. This is because we used the cache only to perform the initial binary search, and query latencies were dominated by the sub-tree traversal. That said, it is easy to extend the lookup cache to apply to sub-tree traversal as well, and thus help in improving the performance of queries.

5.4 Effect of Linearization

Our PHT implementation uses z-curve linearization to convert multi-dimensional indexes into a single dimension. As an exercise, we compared how this linearization technique compares with two other techniques, Hilbert curves and Gray-coded curves [16]. The results of these experiments for an input data set of 100,000 items and block size of 500 are summarized in Table 2. Although Hilbert curves are theoretically shown to have better clustering properties than z-curves, for two-dimensional queries the benefits are limited. Moreover, the advantage of Hilbert curves in producing linearizations with fewer non-contiguous segments while resolving a query is not much of an issue for PHTs since the entire query is processed in parallel starting at the top of a sub-tree of the PHT (unlike in the case of disk indexes where discontinuity implies additional disk seeks).

Although we only experimented with PHTs for two dimensions, they can be extended to an arbitrary number of dimensions. For high-dimensional data, more complex linearizations such as the Pyramid-Technique [5] are known to perform better. With some effort, it should be possible to adapt this linearization to use in conjunction with PHTs.

5.5 Handling concurrency

As we mentioned in Section 4.5, concurrent PHT operations can result in sub-optimal performance in the absence of concurrency primitives in the DHT. In particular, we notice three behaviors:

- Multiple clients simultaneously split a full leaf node.
- PHT leaf nodes fill up to larger than their block size (because multiple clients attempt to insert an item into the node at the same time).
- Insertions are lost when for instance two clients simultaneously attempt to fill the last available slot in a leaf node, one client succeeds, a third client then splits the leaf node, and while that is in progress the second client's insert is lost.

Linearization	Avg depth	Block occupancy	Average # of GETs per query
Z-curve	18.417	39.526	26.21
Hilbert curve	18.424	39.063	25.76
Gray code	18.42	39.526	26.206

Table 2: *Variation in PHT characteristics for different linearization types.*

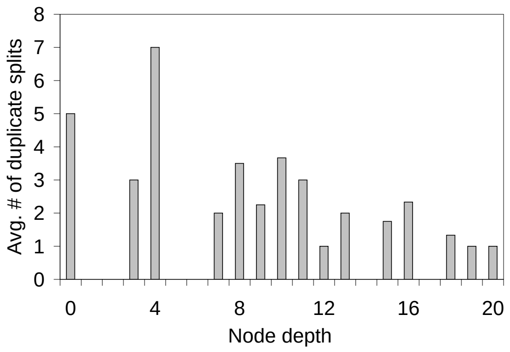


Figure 12: *The average number of duplicate splits as a function of node depth for 25 concurrent PHT writers.*

We measured the frequency with which these behaviors occur with concurrent operations. We ran an experiment with 25 concurrent clients inserting data into a PHT (starting with an empty PHT). Figure 12 shows a plot of the average number of duplicate splits that occur at each node depth within the PHT. We note that contention happens more often closer to the root of the tree. As the tree grows, the number of unique leaf nodes increases and consequently, race conditions for the same leaf decrease. We saw similar behavior for the other two cases.

We then upgraded our DHT deployment to include support for the atomic test-and-set operation and re-ran the above experiments. With this simple addition to the DHT APIs, the PHT was able to operate correctly and no longer exhibited any of the behaviors described above. One should note though that even in the absence of concurrency primitives in the underlying DHT, the above problems either only cause fleeting inefficiencies in the operation of the PHT or can be repaired by the refresh mechanisms.

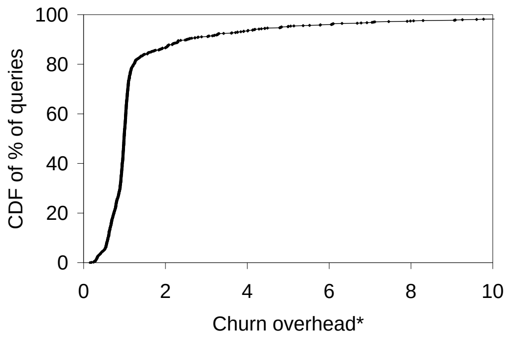


Figure 13: *A CDF of the percentage of queries as a function of the churn overhead. *Churn overhead is defined as the ratio of the query response time with churn versus response time without churn.*

5.6 Dealing with churn

To evaluate the efficacy of the DHT for handling issues of robustness, availability, and replication, we performed a set of experiments where we introduced churn in the DHT. Over one-minute intervals, we randomly killed an existing DHT node or started a new node. For a DHT service (as opposed to a client-based P2P system), this is admittedly a rather high churn rate. We measured the effect of the churn on a query workload with respect to the percentage of expected query responses that were lost due to churn. Our results indicate that there is negligible loss in query responses. Only 2.5% of the queries reported fewer results than expected. Amongst these queries, most still reported over 75% of the expected results. Only in two cases was the loss greater, and this was because the total expected number of results was quite small (fewer than 80 items). Moreover, the data loss was temporary and was recovered as soon as the DHT replication and recovery algorithms kicked in.

We also measured the latency overhead introduced as a result of the churn. We define the churn overhead as the ratio of the query response time with churn versus the response time without churn. Figure 13 plots a CDF of the percentage of queries as a function of the churn overhead. In spite of the churn in the system, most queries show negligible overhead, and only a small number of queries are affected significantly and take much longer to respond. The overhead is largely due to momentary increases in DHT routing latency and replication overhead. Most of the queries that reported fewer than expected items were exactly the ones that had amongst the highest overhead. (On the other hand, some queries performed faster under churn, that is largely an effect of the vagaries of Internet latencies.)

Finally, the true evaluation of the effect of churn is how it affects the end-user application. We measured the effect of data loss in the PHT (due to large amounts of churn) on the accuracy of client device location using Place Lab. We used a simple positioning algorithm proposed for Place Lab that computes a Venn-diagram-like intersection of the observed beacons based on a simple radio propagation model [19]. Even under catastrophic failure that causes significant loss of

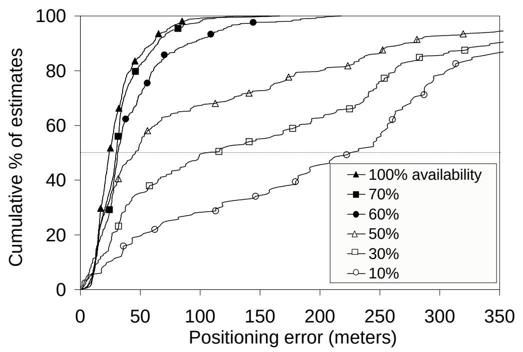


Figure 14: *CDFs of positioning error for varying availability of mapping data.*

Place Lab's mapping data, the application is resilient enough to be able to handle this loss. Figure 14 shows CDFs of the error in positioning of the client device for a range of available mapping data measured in the downtown Seattle area. We note that even with a drop of availability down to as low as 70% (that is, 30% missing data), we see negligible effect on positioning error. However, as availability goes lower than this point, the positioning error starts to increase. Thus, churn is not a significant problem for Place Lab. Not only does the DHT hide most of the effects of churn from Place Lab, but also when data does get lost, Place Lab is capable of absorbing the effects of that loss with minimal observable effects for the user.

6 Lessons Learned

Our experience with building Place Lab on top of OpenDHT demonstrates that it is indeed feasible to build applications with more complex semantics than simply put/get entirely on top of a third-party DHT. Below, we summarize some of the lessons drawn from this experience.

Simplicity of implementation

The code required to hook Place Lab into the underlying OpenDHT service including the entire PHT implementation consists of 2100 lines of Java. In comparison, the OpenDHT codebase is over 14000 lines of code. A customized non-layered implementation would have required Place Lab to implement from scratch all of the scalable routing, robustness, and management properties that we get from OpenDHT for free.

A number of features of Place Lab made it well-suited for this strictly layered implementation. Its data structures are link-free; although the PHT is a tree structure, no explicit links need to be maintained between its nodes. This makes each node largely independent of the others and hence can be easily distributed on top of the DHT. Similarly, information for each Place Lab beacon is mostly independent of the other beacons, thus making it easy to decompose the data across servers. Moreover, Place Lab's mapping data has significant redundancy. This allows it to mask transient failures effectively. Finally, the data structures are capable of refreshing themselves and recovering from failures. This makes them well-suited to deployment over an infrastructure that the Place Lab developers have no control over.

Ease of deployment

We started this discussion by asking the question of whether building Place Lab's mapping service on top of OpenDHT simplifies its deployment. This question has two facets: long-term service deployment, and experimental deployment for performance testing. As a long-term deployment strategy, our implementation of Place Lab is able to hand off much of the management overhead of running and maintaining the distributed system to OpenDHT. Each mapping server in Place Lab is essentially independent of all other servers. OpenDHT provides all of the communication, storage, and replication primitives necessary to support Place Lab's distributed operations. Even if a mapping server dies or is removed from the system, OpenDHT takes care of re-organizing the system around this failure. By outsourcing the OpenDHT deployment to a third-party, a participant in the Place Lab infrastructure only has worry about the management of their individual mapping servers and their connection to the DHT.

On the other hand, while experimenting with the application and its performance, we still ended up having to install our own OpenDHT infrastructure (separate from the existing deployed version). Since OpenDHT is a shared service, the maintainers of the service were unwilling to kill machines at random to allow us to experiment with the effects of churn. Similarly, as we discuss below, we extended the OpenDHT APIs, which would have also resulted in a disruption of the shared OpenDHT deployment if the maintainers were to upgrade it to support the new APIs. To a large extent, this is no different from experimenting with, say, Internet protocols where one cannot expect to tinker directly with the deployed shared infrastructure.

Flexibility of APIs

We were able to build Place Lab entirely on top of a narrow set of application-independent APIs. Our experience demonstrated that although put and get were the primary interfaces that Place Lab relied on, it needed additional auxiliary APIs to correctly and efficiently support the distributed data structures for its application. In particular, we added two features to the underlying DHT: remove(), a mechanism to explicitly invalidate content in the DHT, and put_conditional(), a general-purpose test-and-set operation. However, neither of these modifications is specific to Place Lab. They can be valuable to many other distributed DHT-based applications, and it would be reasonable to expect these features from a deployed DHT service. Our feedback based on this experience has helped the OpenDHT developers refine their APIs to expand the range of applications that they can support.

DHT Guarantees

Typically, DHTs are designed for "best-effort" performance. They provide no concurrency primitives nor do they provide any atomicity guarantees for reads and writes. Although this may be sufficient for simple rendezvous and storage applications, they make it difficult to build more complex data structures whose operations may require atomic sequences of put/get calls. This principle of simplicity over strong guarantees is a central tenet of the design of the Internet. Obviously, it has served the Internet well, but it is less clear whether it is sufficient for large applications over the Internet. PHTs overcome this by using TTLs and periodic refresh to recover from concurrency problems, but even a simple test-and-set operation that provides more than best-effort guarantees goes a long way to improving PHT performance.

Performance

Although the use of DHTs to implement Place Lab's distributed infrastructure significantly simplified its implementation and deployment, this was at the expense of performance. Queries take on average 2–4 seconds (depending on the size of the input data set). In contrast, a single centralized implementation would eliminate the many round trips that account for the performance overhead. Similarly, an implementation that allowed for modifying the underlying DHT routing (for example, Mercury) can also provide opportunities for optimization. This tradeoff is inherent in any layered versus monolithic implementation.

Aggressive caching significantly improves Place Lab's performance. For example, if the PHT data structure is modified infrequently, we can eliminate many of the round trips by caching what amounts to a representation of the current shape of the tree. Applications that can use such forms of caching will be well-suited to provide reasonable performance. Of course, in the end, whether the performance tradeoff is worth the ease of implementation and deployment depends entirely on the requirements of the application and its users.

7 Conclusion

In this paper, we have explored the viability of a DHT service as a general purpose building block for Place Lab, an end-user positioning system. In particular, we investigated the suitability of *layering* Place Lab entirely on top of a third-party DHT to minimize its deployment and management overhead. Place Lab differs from many traditional DHT applications in that it requires stronger semantics than simply put/get operations; specifically, it needs two-dimensional geographic range queries. Hence, we designed and evaluated Prefix Hash Trees, a multi-dimensional range query data structure layered on top of the OpenDHT service. PHTs provide an elegant solution for gathering radio beacon data by location from across the many Place Lab mapping servers.

The layered approach to building Place Lab allowed us to automatically inherit the robustness, availability, and scalable routing properties of the DHT. Although we were able to significantly reduce the implementation overhead by layering, this simplification was at the price of performance. Place Lab and PHTs are unable to make use of optimizations that would have been possible if one were to use a customized DHT underneath.

This is certainly not the last word on the feasibility of general-purpose DHTs as a building block for large-scale applications. However, Place Lab demonstrates that if ease of deployment is a primary criterion (over maximal efficiency), the simple DHT APIs (with minor extensions) can provide the necessary primitives to build richer more complex systems on top.

References

[1] Aberer, K. P-Grid: A self-organizing access structure for P2P information systems. In *Proc. CoopIS* (2001).

[2] Anonymous, et al. OpenDHT: A public DHT service and its uses. Under submission to SIGCOMM 2005.

[3] Aspnes, J., Kirsch, J., and Krishnamurthy, A. Load balancing and locality in range-queriable data structures. In *Proceedings of the Twenty-Third ACM Symposium on Principles of Distributed Computing* (July 2004).

[4] Aspnes, J., and Shah, G. Skip graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2003).

[5] Berchtold, S., Ohm, C. B., and Kriegel, H.-P. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of International Conference on Management of Data (SIGMOD '98)* (June 1998).

[6] Bhagwan, R., Varghese, G., and Voelker, G. Cone: Augmenting DHTs to support distributed resource discovery. Tech. Rep. CS2003-0755, UC, SanDiego, Jul 2003.

[7] Bharambe, A. R., Agrawal, M., and Seshan, S. Mercury: Supporting scalable multi-attribute range queries. In *Proc. SIGCOMM* (2004).

[8] Crainiceanu, A., Linga, P., Gehrke, J., and Shanmugasundaram, J. Querying Peer-to-Peer Networks Using P-Trees. In *Proc. WebDB Workshop* (2004).

[9] Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (Lake Louise, AB, Canada, October 2001).

[10] Druschel, P., and Rowstron, A. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (Lake Louise, AB, Canada, October 2001).

[11] FIPS PUB 180-1. Secure Hash Standard, April 1995.

[12] Freedman, M. J., Freudenthal, E., and Mazières, D. Democratizing Content Publication with Coral. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI 2004)* (San Francisco, Mar. 2004).

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

[14] Gupta, A., Agrawal, D., and Abbad, A. E. Approximate range selection queries in peer-to-peer systems. In *Proc.Conference on Innovative Data Systems Research (CIDR)* (2003).

[15] Huebsch, R., Chun, B., Hellerstein, J. M., Loo, B. T., Maniatis, P., Roscoe, T., Shenker, S., Stoica, I., and Yumerefendi, A. R. The Architecture of PIER: an Internet-Scale Query Processor. In *Proc. Conference on Innovative Data Systems Research (CIDR)* (Jan. 2005).

[16] Jagadish, H. V. Linear clustering of objects with multiple attributes. In *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD'90)* (May 1990), pp. 332–342.

[17] Karger, D. R., and Ruhl, M. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proc. SPAA* (2004).

[18] Kubiatowicz, J. Oceanstore: An Architecture for Global-Scalable Persistent Storage. In *Proceedings of the ASPLOS 2000* (Cambridge, MA, USA, November 2000).

[19] LaMarca, A., et al. Place Lab: Device Positioning Using Radio Beacons in the Wild, May 2005. To appear in Proceedings of Pervasive 2005.

[20] Lamport, L. The Part-Time Parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[21] Mislove, A., Post, A., Reis, C., Willmann, P., Druschel, P., Wallach, D. S., Bonnaire, X., Sens, P., Busca, J.-M., and Arantes-Bezerra, L. POST: A secure, resilient, cooperative messaging system. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems* (Lihue, HI, May 2003).

[22] Muthitacharoen, A., Gilbert, S., and Morris, R. Etna: a fault-tolerant algorithm for atomic mutable dht data. Technical report, Massachussetts Institute of Technology, June 2004.

[23] Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. Ivy: A read/write peer-to-peer file system. In *Proc. OSDI* (2002).

[24] Peterson, L., Anderson, T., Culler, D., and Roscoe, T. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the ACM HotNets-I Workshop* (Princeton, NJ, Oct. 2002). See also http://www.planet-lab.org/.

[25] Rowstron, A., Kermarrec, A.-M., Castro, M., and Druschel, P. Scribe: The design of a large-scale event notification infrastructure. In *Networked Group Communication, Third International COST264 Workshop (NGC'2001)* (Nov. 2001), J. Crowcroft and M. Hofmann, Eds., vol. 2233 of *Lecture Notes in Computer Science*, pp. 30–43.

[26] Sit, E., Dabek, F., and Robertson, J. UsenetDHT: A low overhead usenet server. In *Proc. of the 3rd IPTPS* (Feb. 2004).

[27] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *Proceedings of the ACM SIGCOMM 2002* (Pittsburgh, PA, USA, August 2002).

[28] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM 2001* (San Diego, CA, USA, August 2001).

[29] Tang, C., Xu, Z., and Mahalingam, M. pSearch: Information Retrieval in Structured Overlays. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003), 89–94.

[30] Yalagandula, P., and Browne, J. Solving range queries in a distributed system. Tech. Rep. TR-04-18, UT CS, 2003.

[31] Yalagandula, P., and Dahlin, M. A scalable distributed information management system. In *Proc. SIGCOMM* (2004).