Privacy-preserving, location-based queries using DHTs and R-Trees

Giovanni Campagna, Keshav Santhanam

Abstract

Social running applications are currently unable to query and process locations in large-scale, churn-heavy environments while also respecting the privacy of their users. To address this issue we present hDHT, a system that combines a distributed hash table (DHT) with Hilbert R-Trees to enable location-based queries in a distributed setting. By limiting the resolution of the R-Tree, hDHT is able to obfuscate the true locations of users in the system, thus enabling stronger privacy conditions. In this paper we discuss the novel routing strategy of hDHT as well as its C++ implementation.

I. Introduction

Mobile applications such as Strava [1] provide an easy way for runners to track their activity and share it with friends. However, these applications lack the ability to pair together runners who share similar routes. This task is not trivial as there are several constraints that must be met in order for such an application to be widely adopted:

- Users of the application must be able to share their routes with other users
- The application should be able to compute similar routes quickly
- Users should not be able to see the exact location of any other user

To satisfy these constraints we present hDHT, a system that combines the geo-distributed key-value lookup capabilities of a distributed hash table (DHT) with the fast, location-based queries enabled by Hilbert R-Trees [2].

This paper proceeds as follows. Section II introduces the technologies of DHTs and Hilbert R-Trees. Section III provides a high-level overview of our system architecture. Section IV delves into the implementation details of hDHT. Section V discusses a similar system to hDHT. Section VI enumerates potential next steps for this project and Section VII concludes.

II. Background

In this section we give a brief overview of the core technologies that support hDHT.

A. Distributed hash tables

A distributed hash table (DHT) is a decentralized system that supports efficient, distributed key-value lookups by maintaining a subset of the total keyspace on each node. Nodes forward requests for keys that are outside its own keyspace to other nodes in the system by keeping some information about its peers, each of which is responsible for a different segment of the keyspace. Early efforts to study DHTs include Chord [3], Tapestry [4], and Pastry [5].

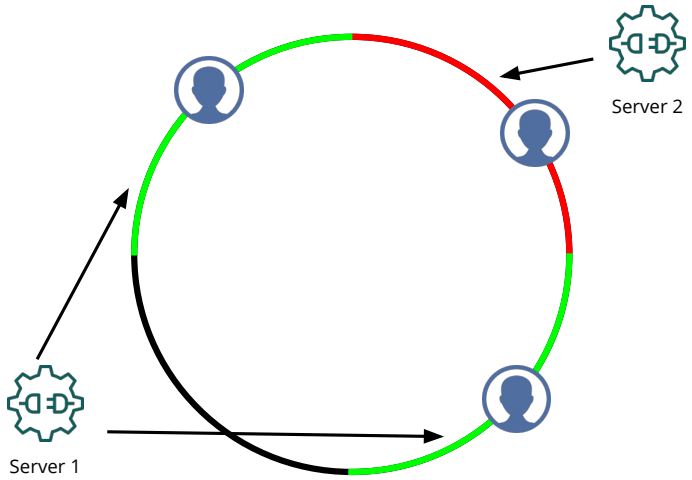

Figure 1: The Architecture of hDHT.

B. Hilbert R-Trees

Kamel and Faloutsos proposed the idea of a Hilbert R-Tree as a means of performing efficient location-based queries [2]. Their key insight was that by using a Hilbert curve to determine a linear ordering of the entries in the tree, the splitting policy in overflow scenarios can be tuned in such a way that the utilization of the tree is close to 100%. Each node in a Hilbert R-Tree keeps track of the largest Hilbert value (LHV) among the entries in its subtree, as well as the maximum bounding rectangle (MBR) over all its subtree entries. Entries are then inserted in order of increasing Hilbert value, and the LHV and MBR of each node are adjusted as necessary after every modification to the tree.

III. System Design

The goal of our system is to store the location, and associated information, of a large number of clients who can potentially move around in the real world. Once the location is stored, our system can serve requests against those clients efficiently, leveraging the physical locality properties. In this section, we introduce the design, while Section IV discusses some tradeoffs in our implementation.

A. Architecture

The architecture of hDHT is shown in Figure 1. In hDHT, the server control one or more contiguous segments on a *consistent hashing ring*. Each client has a position in the hashing ring which is determined based on the client physical location, using the *Hilbert curve function*. Each client thus *registers* with the server that controls the segment containing the client.

In the example, Server 1 controls the two green segments, and stores the data for two clients. Server 2 controls the red segment, and stores the data only of the client in that segment. Other servers, not shown, control the rest the DHT curve.

B. Client Interface

Clients of hDHT are likely to be on a mobile device, as they should follow and record precisely the location of the user. Servers cannot perform RPCs towards the clients, because the clients might be behind a restrictive firewall. For this reason, our design moves most of the computation and networking to the server, and initiates all client-server communication from the client.

- FINDCONTROLLINGSERVER(*node_id*) $\rightarrow$ *address*: find the address of the server controlling the given node ID; this request succeeds even if the node ID does not correspond to any client registered with the server
- FINDSERVERFORPOINT(*position*) $\rightarrow$ *address*: find the address of the server controlling the given position
- CLIENTHELLO(*position*, *node_id*) $\rightarrow$ *ok*, *node_id*: register the client against this server; the client transmits both its own position and the previously obtained node ID, if available; the server responds *Created* if the client was newly registered, *AlreadyExists* if it recognized the previous node ID, or *WrongServer* if the server is not responsible for the region of the Hilbert curve corresponding to *position*.
- MOVE(*position*) $\rightarrow$ *ok*, *node_id*, *address*: informs the server that the client moved; the server replies with the new node ID for the client, and either *SameServer* or *DifferentServer*, to indicate what server is now responsible for the client; if the client receives *DifferentServer*, it must register with the new server
- SETMETADATA(*key*, *value*): set metadata for the calling client
- GETMETADATA(*node_id*, *key*) $\rightarrow$ *value*: get metadata about the given client; the server forwards the request to the controlling server if needed

Figure 2: The hDHT client-server protocol

Each client is identified by a 160-bit node identifier. This identifier is opaque to the client. The client additionally stores its own position, as obtained from the GPS senstor, and the address of the server it is registered with. The client interacts primarily with the registering server, but any time it can retrieve the address of the server responsible for any point in the curve.

Servers store simple client metadata, in the form of string key-value pairs. Each server serves metadata requests only for the clients it is responsible for, which ensures consistency.

The full client-server interface is shown in Figure 2.

C. Hilbert Curves

Client node IDs define the position of a client in the consistent hashing ring. To facilitate location based queries, the node IDs are generated using a *locality sensitive hash* that uses a Hilbert curve.

A Hilbert curve, or *space-filling curve*, is a curve that passes all points in a grid exactly once and does not self intersect. The Hilbert curve can be constructed for grids of arbitrary dimension. The Hilbert curve of order k, which covers the grid of size $2^k \times 2^k$, is constructed as a fractal, by refining the Hilbert curve of order $k-1$. An efficient $O(k)$ algorithm exists to convert any point to the Hilbert curve to its coordinates in the grid and viceversa.

D. Mapping Hilbert Curves to Consistent Hashing

hDHT uses the Hilbert curve concept to generate the ID of the client. First, the physical position of the client is mapped to a point in a rectangular Earth-sized grid. Then, the point in the grid is converted to a point in the Hilbert curve, which provides the high bits of the node IDs. The remaining low bits are assigned randomly. This is a locality sensitive hashing because clients who are phyisically close will have similar Hilbert curve values, and thus will share some of the high bits of their node IDs. Additionally, clients with similar high bits in the node IDs are more likely to be assigned to the same server, which improves the efficiency of serving requests.

Furthermore, hDHT must partition the grid in a way that each server controls a rectangular region. This way, search requests can be served efficiently by dividing the query into disjoint rectangles. To do so, the Hilbert curve must be partitioned into *intervals* of the form:

$$\left[k2^i, (k+1)2^i\right) \tag{1}$$

(with k, i integers, and i less than the order of the curve). Intervals of this form can be shown to be non-overlapping in the grid.

Each server controls one or more of these intervals. The server has complete information over the intervals it controls, and relies on other servers for the rest of the grid. For each controlled interval of the Hilbert curve, the server maintains an *Hilbert R-Tree* [2] of the clients. This is a data structure that allows efficient insertion, deletion and query by rectangle.

E. Performing Search

To perform a search, the server needs to efficiently identify which intervals of the DHT are partially covered by the search rectangle, and then perform the search within the interval.

The algorithm takes advantage of an important property of Hilbert curves: if a point is outside a rectangle in the Hilbert curve, all points on the curve between that point and the next rectangle corner are also outside the rectangle.

The server then uses the following algorithm:

Algorithm 1 hDHT Routing

query $\leftarrow$ search rectangle
Compute h_0, h_1, h_2, h_3 of the 4 corners of *query* $\triangleright$ h_0 is the smallest Hilbert value, h_3 is the largest
current_point $\leftarrow h_0$
while *current_point* $\leq h_3$ **do**
    if *query* contains *current_point* **then**
        Retrieve interval I containing *current_point* from DHT
        if I is local interval **then**
            Use local R-Tree to perform search
        else
            Forward search to the server that contains I
            Merge the results
        end if
        current_point $\leftarrow h', h' = (k+1)2^i$
    else
        current_point $\leftarrow \{\min\ h_i \mid h_i >$ *current_point*$\}$
    end if
end while

The complexity of the algorithm is linear in the number of intervals spanned by the query rectangle. The server keeps this number low by merging adjacent local intervals.

F. Routing and Load Balancing

To make it possible for servers to join and leave the DHT easily, each server does not keep precise information on which intervals are owned by whom.

At initialization time, the server starts with a single peer, which it assumes controls the whole table. The server registers with this initial peer, who responds with a list of interval it knows about. The server then registers with each of the newly discovered peers, which will further send more and more precise information.

Additionally, when a new server joins the DHT, each peers that learns about the new server performs load balancing on the portion of the table it controls, by spilling the intervals it controls and transferring control of the newly created subintervals to the new server. Intervals must be split in half at each step, because they must be power-of-2 sized to be rectangular in the grid.

An interval will be considered for splitting if it is larger than a threshold (in logarithmic size), or if the number of clients exceeds a threshold. In the former case, the interval is split in half only once, and

one half is sent to the new server. In the latter case, the newly created half interval with the most clients is split again until the number of clients is below the threshold. Then, the half interval with the least clients is sent to the new server.

Servers automatically move the metadata and client IDs during DHT rebalancing, but they do not inform clients. Clients learn of the rebalancing by receiving a redirection error the next time they attempt to perform a request.

Under this model, a request can be routed to the corresponding server in at most $O(\log n)$ steps, where $n = 2^k$ is the number of points in the grid. This is because at every step the interval is split at least in half. In the common case of a fully populated table where each server has interval of roughly the same size this scheme is logarithmic also in the number of nodes, and thus optimal.

IV. Implementation

hDHT was implemented as a self contained library, as well as a daemon to act as the server. The hDHT library also includes an example command line client, which allows one to test the capabilities of the system without physically moving around.

hDHT is open source, and available on Github [1]. We implemented hDHT in approximately 5,000 lines of C++ code.

A. Library Design

hDHT uses C++, and the libuv library [2] for the event loop and asynchronous portable IO. It has been tested on Linux and OS X.

The client API of the library consists of a single `Client` object, which exposes methods to get and set the current location, get and set the local metadata, and perform queries against remote clients. All other objects in the library, including the R-Tree implementation and the DHT abstraction, are hidden from library users.

To aid in code reuse, and reduce disk footprint, the library also includes the full server implementation. The server binary is a shim that initializes the library in server mode.

B. R-Tree

We built a self-contained R-Tree library from scratch by using the algorithms provided in the Hilbert R-Tree paper [2]. The `RTree` class provides a minimal interface to allow for inserting data into the tree at a particular `Point` in the 2D space and performing a search using a `Rectangle`. The search returns a list of `NodeID`s that are located within the query rectangle.

C. Peer Discovery and Connectivity Requirements

In the current implementation, all peers are expected to be reachable by a direct TCP connection at least in one direction.

Peers are specified by address (IP and port) on the command line of the server. If no peer is specified when the server starts, the server assumes control of the whole table, and waits for incoming connections. All servers also listen for incoming connections on a specified IP and port, and publish this address to other servers.

[1] https://github.com/gcampax/hDHT

[2] http://libuv.org

D. RPC Library

To keep the library small, and avoid dependencies (especially dependencies that the authors were not familiar with when the project was started), hDHT uses a small home-grown RPC library, over TCP sockets. The library uses a pseudo-IDL defined using the C++ preprocessor, which makes it avoid a separate compilation step, and makes it easy to integrate with IDEs and build systems. It then makes use of C++ template metaprogramming to generate safe and efficient marshalling code. All RPC requests in the library are asynchronous; the caller passes a callback to be notified of the completion of the RPC.

V. Related Work

The project most similar to hDHT is the Content Addressable Network (CAN) proposed by Ratnasamy, et al [6]. The primary difference between CAN and hDHT is that with a dimensionality of 2, CAN gives an average case routing path length of $O(\sqrt{n})$ where n is the number of zones, whereas hDHT guarantees a routing path length of $O(\log n)$ as discussed in Section III.

VI. Future Work

Now that we have implemented the underlying infrastructure of hDHT, the next step would be to write a client that actually sits on a mobile device and has access to real location data.

Additionally, we would like to develop an algorithm that could take as input the information about runners' locations and output similar routes. One example of such an algorithm could look at clients that sit in the same RTree node for a long period time; if the clients are in the same RTree node after they move, it means they move relatively close to each other. Another possibility for the algorithm would be to consider each segment or portion of the Hilbert curve as a feature in a large collaborative filtering model [7]. This would have the advantage to be the most general recommendation system algorithm, but it would require a training phase with a large dataset.

Finally, we would want to build a more interactive interface so that we could visualize the results of our route similarity algorithm.

VII. Conclusion

We observe that existing running applications are not able to support location-based queries in a distributed environment while maintaining their users' privacy. To this end, we present hDHT, a system that uses a distributed hash table with an efficient routing scheme enabled by Hilbert R-Trees to facilitate such queries. hDHT splits the Hilbert curve into several distinct intervals and assigns repsonsibility for each of the intervals to a node in the DHT. Then users can insert themselves into the system and lookup information about other nodes using routing paths of length $\mathcal{O}(\log n)$. In addition, users' privacy is preserved by limiting the resolution of the R-Tree. We believe this approach will enable novel applications in the future that take advantage of the fast location-based query capabilities of hDHT.

References

[1] "Strava," https://www.strava.com/, accessed: 2017-12-11.

[2] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," Tech. Rep., 1993.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[4] B. Y. Zhao, J. Kubiatowicz, A. D. Joseph *et al.*, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," 2001.

[5] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2001, pp. 329–350.

[6] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*. ACM, 2001, vol. 31, no. 4.

[7] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01. New York, NY, USA: ACM, 2001, pp. 285–295. [Online]. Available: http://doi.acm.org/10.1145/371920.372071