UrDHT: A Unified Model for Distributed Hash Tables

Andrew Rosen Brendan Benshoof Robert W. Harrison Anu G. Bourgeois
Department of Computer Science
Georgia State University
Atlanta, Georgia
rosen@cs.gsu.edu bbenshoof@cs.gsu.edu rharrison@cs.gsu.edu anu@cs.gsu.edu

***Abstract*—Distributed Hash Tables (DHTs) have an inherent set of qualities, such as greedy routing, maintaining lists of peers which define the topology, and form an overlay network. Rather than having a developer be concerned with the details of a given DHT, we have constructed a new framework, UrDHT, that generalizes the functionality and implementation of various DHTs.**

UrDHT is an abstract model of a Distributed Hash Table that implements a self-organizing web of computational units. It maps the topologies of DHTs to the primal-dual problem of Voronoi Tessellation and Delaunay Triangulation. By completing a few simple functions, a developer can implement the topology of any DHT in any arbitrary space using UrDHT. For example, we implemented a DHT operating in a hyperbolic geometry, a previously unexplored nontrivial metric space with potential applications, such as latency embedding.

***Index Terms*—Peer-to-Peer Networks; Distributed Hash Tables; Computational Geometry; Delaunay Triangulation; Voronoi Tessellation; Self-Organizing Networks;**

I. Introduction

UrDHT is an abstract model of a distributed hash table (DHT). It is a unified and cohesive model for creating DHTs and P2P applications based on DHTs.

Distributed Hash Tables have been the catalyst for the creation of many P2P applications. Among these are Redis [1], Freenet [2], and, most notably, BitTorrent [3]. All DHTs use functionally similar protocols to perform lookup, storage, and retrieval operations. Despite this, no one has created a cohesive formal DHT specification.

Our primary motivation for this project was to create an abstracted model for Distributed Hash Tables based on observations we made during previous research [4]. We found that all DHTs can cleanly be mapped to the primal-dual problems of Voronoi Tessellation and Delaunay Triangulation.

UrDHT builds its topology directly upon this insight. It uses a greedy distributed heuristic for approximating Delaunay Triangulations. We found that we could reproduce the topology of different DHTs by defining a selection heuristic and rejection algorithm for the geometry the DHT. For every DHT we implemented, our greedy approximation of Delaunay Triangulation produced a stable DHT, regardless of the geometry. This works in non-Euclidean geometries such as XOR (Kademlia) or even a hyperbolic geometry represented by a Poincarè disc.

The end result is not only do we have an abstract model of DHTs, we have a simple framework that developers can use to quickly create new distributed applications. This simple framework allows generation of internally consistent implementations of different DHTs that can have their performance rigorously compared.

To summarize our contributions:

- We give a formal specification for what needs to be defined in order to create a functioning DHT. While there has long existed a well known protocol shared by distributed hash tables, this defines what a DHT does. It does not describe what a DHT is.
  We show that DHTs cleanly map to the primal-dual problem of Delaunay Triangulation and Voronoi Tessellation. We list a set of simple functions that, once defined, allow our Distributed Greedy Voronoi Heuristic (DGVH) to be run in any space, creating a DHT overlay for that space (Section II).
- We present UrDHT as an abstract DHT and show how a developer would modify the functions we defined to create an arbitrary new DHT topology (Section III).
- We show how to reproduce the topology of Chord and Kademlia using UrDHT. We also implement a DHT in a Euclidean geometry and a hyperbolic geometry represented by a Poincarè disc (Section IV).
- We conduct experiments that show building

DHTs using UrDHT produced efficiently routable networks, regardless of the underlying geometry(Section V).
- We discuss the ramifications of our work and what future work is available (Section VI).

II. What Defines a DHT

A distributed hash table is usually defined by its protocol; in other words, what it can do. Nodes and data in a DHT are assigned unique[1] keys via a consistent hashing algorithm. To make it easier to intuitively understand the context, we will call the key associated with a node its ID and refer to nodes and their IDs interchangeably.

A DHT can perform the `lookup(key)`, `get(key)`, and `store(key, value)` operations.[2] The `lookup` operation returns the node responsible for a queried key. The `store` function stores that key/value pair in the DHT, while `get` returns the value associated with that key.

However, these operations define the functionality of a DHT, but do not define the requirements for implementation. We define the necessary components that comprise DHTs. We show that these components are essentially Voronoi Tessellation and Delaunay Triangulation.

A. DHTs, Delaunay Triangulation, and Voronoi Tessellation

Nodes in different DHTs have, what appears at the first glance, wildly disparate ways of keeping track of peers - the other nodes in the network. However, peers can be split into two groups.

The first group is the *short peers*. These are the closest peers to the node and define the range of keys the node is responsible for. A node is responsible for a key if and only if its ID is closest to the given key in the geometry of the DHT. Short peers define the DHTs topology and guarantee that the greedy routing algorithm shared by all DHTs works.

Long peers are the nodes that allow a DHT to achieve faster routing speeds than the topology would allow using only short peers. This is typically $O(\log(n))$ hops, although polylogarithmic time is acceptable [5]. A DHT can still function without long peers.

Interestingly, despite the diversity of DHT topologies and how each DHT organizes short and long peers, all DHTs use functionally identical greedy routing algorithms (Algorithm 1):

[1]Unique with astronomically high probability, given a large enough consistent hashing algorithm.

[2]There is typically a *delete(key)* operation too, but it is not strictly necessary.

Algorithm 1 The DHT Generic Routing algorithm

```
1:  function $n$.LOOKUP(($key$))
2:      if $key \in n$'s range of responsibility then
3:          return $n$
4:      end if
5:      if One of $n$'s short peers is responsible for $key$ then
6:          return the responsible node
7:      end if
8:      $candidates$ = $short\_peers$ + $long\_peers$
9:      $next \leftarrow \min(n.\text{distance}(candidates, key))$
10:     return $next$.lookup($key$)
11: end function
```

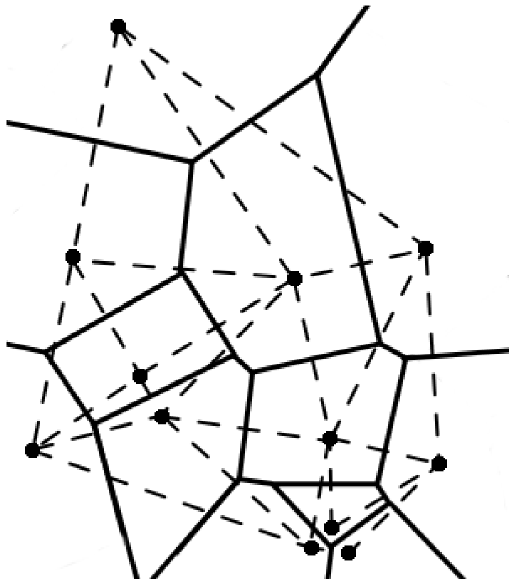

Fig. 1: An example Voronoi diagram for objects on a 2-dimensional space. The black lines correspond to the borders of the Voronoi region, while the dashed lines correspond to the edges of the Delaunay Triangulation.

The algorithm is as follows: If I, the node, am responsible for the key, I return myself. Otherwise, if I know who is responsible for this key, I return that node. Finally, if that is not the case, I forward this query to the node I know with shortest distance from the node to the desired key.[3]

Depending of the specific DHT, this algorithm might be implemented either recursively or iteratively. It will certainly have differences in how a node handles errors, such as how to handle connecting to a node that no longer exists. This algorithm may possibly be run in parallel, such as in Kademlia [6]. The base greedy algorithm is always the same regardless of the implementation.

[3]This order matters, as some DHTs such as Chord are unidirectional.

With the components of a DHT defined above, we can now show the relationship between DHTs and the primal-dual problems of Delaunay Triangulation and Voronoi Tessellation. An example Delaunay Triangulation and Voronoi Tessellation is show in Figure 1.

We can map a given node's ID to a point in a space and the set of short peers to the Delaunay Triangulation. This would make the range of keys a node is responsible correspond to the node's Voronoi region. Long peers serve as shortcuts across the mesh formed by Delaunay Triangulation.

Thus, if we can calculate the Delaunay triangulation between nodes in a DHT, we have a generalized means of creating the overlay network. This can be done with any algorithm that calculates the Delaunay Triangulation.

Computing the Delaunay Triangulation and/or the Voronoi Tessellation of a set of points is a well analyzed problem. Many algorithms exist which efficiently compute a Voronoi Tessellation for a given set of points on a plane, such as Fortune's sweep line algorithm [7].

However, DHTs are completed decentralized, with no single node having global knowledge of the topology. Many of the algorithms to compute Delaunay Triangulation and/or Voronoi Tessellation are unsuited to a distributed environment. In addition, the computational cost increases when we move into spaces with greater than two dimensions. In general, finding the Delaunay Triangulation of n points in a space with d dimensions takes $O(n^{\frac{2d-1}{d}})$ time [8].

Is there an algorithm we can use to efficiently calculate Delaunay Triangulations for a distributed system in an arbitrary space? We created an algorithm called the Distributed Greedy Voronoi Heuristic (DGVH), explained below [4].

B. *Distributed Greedy Voronoi Heuristic*

The Distributed Greedy Voronoi Heuristic (DGVH) is an efficient method for nodes to define their individual Voronoi region (Algorithm 2). DVGH selects nearby nodes that would correspond to points connected to it within a Delaunay triangulation. Our previous implementation relied on a midpoint function [4]. We have refined our heuristic to render a midpoint function unnecessary.

The heuristic is described in Algorithm 2. Every maintenance cycle, nodes exchange their peer lists with their short peers. A node creates a list of candidates by combining their peer lists with their neighbor's peer lists.[4] Sort the list of peers from closest to furthest distance. The node then initializes a new peer list with the closest candidate. For each of the remaining candidates, the node compares the distance between the current short peers and the candidate. If the new peer list does not contain any short peers closer to the candidate than the node, the candidate is added to the new peer list. Otherwise, the candidate is set aside.

The resulting short peers are a subset of the node's actual Delaunay neighbors. A crucial feature is that this subset guarantees that DGVH will form a routable mesh.

Algorithm 2 Distributed Greedy Voronoi Heuristic

```
1: Given node $n$ and its list of $candidates$.
2: Given the minimum $table\_size$
3: $short\_peers \leftarrow$ empty set
4: $long\_peers \leftarrow$ empty set
5: Sort $candidates$ in ascending order by each node's distance to $n$
6: Remove the first member of $candidates$ and add it to $short\_peers$
7: for all $c$ in $candidates$ do
8:     if any node in $short\_peers$ is closer to $c$ than $n$ then
9:         Reject $c$ as a peer
10:    else
11:        Remove $c$ from $candidates$
12:        Add $c$ to $short\_peers$
13:    end if
14: end for
15: while $|short\_peers| < table\_size$ and $|candidates| > 0$ do
16:     Remove the first entry $c$ from $candidates$
17:     Add $c$ to $short\_peers$
18: end while
19: Add $candidates$ to the set of $long\_peers$
20: handleLongPeers($long\_peers$)
```

Candidates are gathered via a gossip protocol as well as notifications from close peers. How long peers are handled depends on the particular DHT implementation. This process is described more in Section III-A.

The expected maximum size of *candidates* corresponds to the expected maximum degree of a vertex in a Delaunay Triangulation. This is $\Theta(\frac{\log n}{\log \log n})$, regardless of the number of the dimensions [9]. We can therefore expect *short peers* to be bounded by $\Theta(\frac{\log n}{\log \log n})$.

The expected worst case cost of DGVH is $O(\frac{\log^4 n}{\log^4 \log n})$

[4]In our previous paper, nodes exchange short peer lists with a single peer. Calls to DGVH in this paper use the information from all their short peers.

[4], regardless of the dimension [4].[5] In most cases, this cost much lower, on the order of $O(dk\log(k)+k^2)$ for d dimensions and k candidates. Additional details can be found in our previous work [4].

We have tested DGVH on Chord (a ring-based topology), Kademlia (an XOR-based tree topology), general Euclidean spaces, and even in a hyperbolic geometry. This is interesting because not only can we implement the contrived topologies of existing DHTs, but more generalizable topologies like Euclidean or hyperbolic geometries. We show in Section V that DGVH works in all of these spaces. DGVH only needs the following functions to be defined in order for nodes to perform lookup operations and determine responsibility.

- **A `distance` function** - This measures distance in the overlay formed by the Distributed Hash Table. In most DHTs, the distance in the overlay has no correlation with real-world attributes.
- **A `responsibility` definition** This defines the range of keys a node is responsible for. Not every DHT defines which node is responsible for particular keys in the same way. For example, nodes in Kademlia are responsible for the keys closest to themselves, while in Chord, nodes are responsible for the keys falling between themselves and the preceding node.

We will now show how we used this information and algorithms to create UrDHT, our abstract model for distributed hash tables.

III. URDHT

The name of UrDHT comes from the the German prefix *ur*, which means "original." Our name states that all DHTs can spring from UrDHT.

UrDHT is divided into 3 broad components: Storage, Networking, and Logic. Storage handles file storage and Networking dictates the protocol for how nodes communicate. These components oversee the lower level mechanics of how files are stored on the network and how bits are transmitted through the network. The specifics are outside the scope of the paper, but can be found on the UrDHT Project site [10].

Most of our discussion will focus on the Logic component. The Logic component is what dictates the behavior of nodes within the DHT and the construction of the overlay network. It is composed of two parts: the DHT Protocol and the Space Math.

The DHT Protocol contains the canonical operations that a DHT performs, while the Space Math is what effectively distinguishes one DHT from another. A developer only needs to change the details of the `space math` package in UrDHT to create a new type of DHT. We discuss each in further detail below.

A. The DHT Protocol

The DHT Protocol (`LogicClass.py`) [10] is the shared functionality between every single DHT. It consists of the node's information, the short peer list that defines the overlay, the long peers that make efficient routing possible, and all the functions that use them. There is no need for a developer to change anything in the DHT Protocol, but it can be modified if so desired. The DHT Protocol depends on functions from Space Math in order to perform operations within the specified space.

Many of the function calls should be familiar to anyone who has study DHTs. We will discuss a few new functions we added and the ones that contribute to node maintenance.

The first thing we note is the absence of `lookup`. In our efforts to further abstract DHTs, we have replaced `lookup` using the function `seek`. The `seek` function acts a single step of `lookup`. It returns the closest node to key that the node knows about. Nodes can perform `lookup` by iteratively calling `seek` until it receives the same answer twice.

The `join` operation takes in a set of bootstrap nodes, called *candidates*, rather than a single node. This is part of our plans about how UrDHT can be used as a bootstrap network by providing bootstrapping information for a particular network. We expect nodes that want to join a particular network to be able to query UrDHT and receive a list of nodes that they can use to bootstrap the joining.

The joining node randomly selects one of these *candidates* and finds the "parent" node currently responsible for the space. The joining node then populates its short peers using the "parent" node's short peers. The node uses the parent to populate its short peer list and then makes it aware of its existence using `notify`. Once that has been finished, the joining node starts its maintenance thread.

Maintenance is done via gossip. Each maintenance cycle, the node recalculates its Delaunay (short) peers using its neighbor's peer list and any nodes that have

[5]As mentioned in the previous footnote, if we are exchanging peers with a single neighbor rather than all our neighbors, the cost lowers to $O(\frac{\log^2 n}{\log^2 \log n})$.

notified it since the last maintenance cycle. This is done using DGVH by default, but could be done with some other algorithm. Once those are calculated, the node handles modifying its long peers, as dictated by the `handleLongPeers` function described in Section III-B.

B. The Space Math

The Space Math consists of the functions that define the DHT's topology. It requires a way to generate short peers to form a routable overlay and a way to choose long peers. We use DGVH for generating short peers, which works in every space tested. Space Math requires the following functions when using DGVH:

- The `idToPoint` function takes in a node's ID and any other attributes needed to map an ID onto a point in the space.
  In the vast majority of DHTs, this `idToPoint` function needs nothing more than the ID as input. The ID is directly translated into a large integer and used as a coordinate in a one dimensional space.
- The `distance` function takes in two points, a and b, and outputs the shortest distance from a to b. This distinction matters since distance is not symmetric in every DHT. The prime example of this is Chord, which is a unidirectional toroidal ring.
- The function `getClosest` returns the point closest to *center* from a list of *candidates*, measured by the distance function. Depending on what you want to measure, `getClosest` might measure the distance from *center* to each of the candidates or from each of the candidates to the *center*.
- We use the above functions to implement `getDelaunayPeers`. Given a set of points, the *candidates*, and a center point *centers*, `getDelaunayPeers` calculates a mesh that is made up of the Delaunay peers of *center*.
  We assume that this is done using DGVH, shown by the Python code in Listing 1

Listing 1: `getDelaunayPeers()`

```
def getDelaunayPeers(candidates,center):
        if len(candidates) < 2:
                return candidates
        sortedCandidates = sorted(candidates,
            key=lambda x: distance(x, center))
        peers = [sortedCandidates[0]]
        sortedCandidates = sortedCandidates[1:]
        for c in sortedCandidates:
                accept = True
                for p in peers:
                        if distance(c,p) <
                            distance(c,center):
                                accept = False
                                break
                if accept:
                        peers.append(c)
        return peers
```

- The final function is `handleLongPeers`. `handleLongPeers` takes in a list of *candidates* and a *center*, much like `getDelaunayPeers`, and returns a set of peers to act as routing shortcuts. The implementation of this function should vary greatly from one DHT to another. For example, Symphony [11] and other small-world [12] networks choose long peers using a probability distribution. Chord has a much more structured distribution, with each long peer being increasing powers of 2 distance away from the node [13].

IV. Implementing other DHTs

A. Implementing Chord

Ring topologies are fairly straightforward since they act as are one dimensional Voronoi tessellations, splitting up what is effectively a modular number line among multiple nodes.

Chord uses a unidirectional distance function. Given two integer keys a and b and a maximum value 2^m, the `distance` from a to b in Chord's unidirectional ring is:

$$distance(a,b) = \begin{cases} 2^m + b - a, & \text{if } b - a < 0 \\ b - a, & \text{otherwise} \end{cases}$$

Short peer selection is trivial in chord, so rather than using DGVH for `getDelaunayPeers`, each node chooses from the list of candidates the candidate closest to it (predecessor) and the candidate it is closest to (successor).

Chord's finger (long peer) selection strategy is emulated by `handleLongPeers`. For each of the ith bits in the hash function, we choose a long peer p_i from the candidates such that

$$p_i = getClosest\,(candidates, t_i)$$

where

$$t_i = (n + 2^i) \mod 2^m$$

for the current node n. The `getClosest` function in Chord should return the candidate with the shortest distance from the candidate to the point.

B. Implementing Kademlia

Kademlia uses the exclusive or, or XOR, metric for distance. This metric, while non-euclidean, is perfectly acceptable for calculating distance in Kademlia [6]. For two given keys a and b

$$disance(a, b) = a \oplus b$$

The `getDelaunayPeers` function uses DGVH as normal to choose the short peers for node n. We then used Kademlia's k-bucket strategy [6] for `handleLongPeers`. The remaining candidates are placed into buckets, each capable holding a maximum of k long peers.

To summarize briefly, node n starts with a single bucket containing n, covering long peers for the entire range. When attempting to add a candidate to a bucket already containing k long peers, if the bucket contains node n, the bucket is split into two buckets, each covering half of that buckets range. Further details of how Kademlia k-buckets work can be found in the Kademlia protocol paper [6].

C. ZHT

ZHT [14] leads to an extremely trivial implementation in UrDHT. Unlike other DHTs, ZHT assumes an extremely low rate of churn. It bases this rationale on the fact that tracking $O(n)$ peers in memory is trivial. This indicates the $O(\log n)$ memory requirement for other DHTs is overzealous and not based on a memory limitation. Rather, the primary motivation for keeping a limited subset of the network in memory is more due to the cost of maintenance messages. ZHT shows, that by assuming low rates of churn (and infrequent maintenance messages as a result), having $O(n)$ peers is a viable tactic for faster lookups.

As a result, the topology of ZHT is a clique, with each node having an edge to all other nodes, yielding $O(1)$ lookup times with an $O(n)$ memory cost. The only change that needs to be made is to consider all peer candidates to be short peers.

D. Implementing a DHT in a non-contrived Metric Space

We used a Euclidean geometry as the default space when building UrDHT and DGVH [4]. For two vectors $\vec{a}$ and $\vec{b}$ in d dimensions:

$$distance\left(\vec{a}, \vec{b}\right) = \sqrt{\sum_{i \in d} (a_i - b_i)^2}$$

We implement `getDelaunayPeers` using DGHV and set the minimum number of short peers to $3d + 1$, a value we found through experimentation [4].

Long peers are randomly selected from the left-over candidates after DGVH is performed [4]. The maximum size of long peers is set to $(3d+1)^2$, but it can be lowered or eliminated if desired and maintain $O(\sqrt[d]{n})$ routing time.

Generalized spaces such as Euclidean space allow the assignment of meaning to arbitrary dimension and allow for the potential for efficient querying of a database stored in a DHT.

We have already shown with Kademlia that UrDHT can operate in a non-Euclidean geometry. Another non-euclidean geometry UrDHT can work in is a hyperbolic geometry.

We implemented a DHT within a hyperbolic geometry using a Poincarè disc model. To do this, we implemented `idToPoint` to create a random point in Euclidean space from a uniform distribution. This point is then mapped to a Poincarè disc model to determine the appropriate Delaunay peers. For any two given points a and b in a Euclidean vector space, the `distance` in the Poincarè disc is:

$$distance(a, b) = \operatorname{arcosh}\left(1 + 2\frac{\|a - b\|^2}{(1 - \|a\|^2)(1 - \|b\|^2)}\right)$$

Now that we have a `distance` function, DGVH can be used in `getDelaunayPeers` to generate an approximate Delaunay Triangulation for the space. The `getDelaunayPeers` and `handleLongPeers` functions are otherwise implemented exactly as they were for Euclidean spaces.

Implementing a DHT in hyperbolic geometry has many interesting implications. Of particular note, embedding into hyperbolic spaces allows us to explore accurate embeddings of internode latency into the metric space [15] [16]. This has the potential to allow for minimal latency DHTs.

V. Experiments

We use simulations to test our implementations of DHTs using UrDHT. Using simulations to test the correctness and relative performance of DHTs is standard practice for testing and analyzing DHTs [6] [13] [17] [11] [18] [19].

Our experiments demonstrate that topologies implemented using UrDHT converge to a routable mesh. We demonstrate that lookup operations in these meshes can be done in sublinear time.

Tests k connected randomly populated network vars: k for your k connected network 10, 20 duration what we have works size 100, 500, 1000, 5000 if possible,

Iterative joins, start with one node vars: join rate (ticks per join) ,1 3 final size 100, 500, 1000, 5000 if possible, join method (bootstrapping size) all,

outputs Network diameter at tick avg greedy routing distance greedy routing success rate at tick maximum degree mean degree std deviatation degree Anything we want for degree

Graph max degree vs expected maximimum degree (this is short peers)

A. Kademlia Works

B

B. Performance of Chord on our network module

C. UrDHT Cohesion Euclidean Space

This experiment greatly resembles the one we performed when evaluating DGVH [4].

We've previously shown that higher degrees work [4].

D. Cohesion in hyperbolic space

We showed in worked in DGVH, it should work in a hyperbolic space.

VI. Applications and Future Work

UrDHT is a unified model for DHTs and framework for building distributed applications. We have shown how it possible to use UrDHT to not only implement traditional DHTs such as Chord and Kademlia, but also in much more generalized spaces such as Euclidean and Hyperbolic geometries.

There are numerous routes we can take with our model. Of particular interest are the applications of building a DHT overlay that operates in a hyperbolic geometry.

One of the other features shared by nearly every DHT is that routing works by minimizing the number of hops across the overlay network, with all hops treated as the same length. This is done because it is assumed that DHTs know nothing about the state of actual infrastructure the overlay is built upon.

However, this means that most DHTs could happily route a message from one continent to another and back. This is obviously undesirable, but it is the status quo in DHTs. The reason for this stems from the generation of node IDs in DHTs. Nodes are typically assigned a point in the range of a cryptographic hash function. The ID corresponds to the hash of some identifier or given a point randomly. This is done for purposes of load balancing and fault tolerance.

For future work, we want to see if there is a means of embedding latency into the DHT, while still maintaining the system's fault tolerance. Doing so would mean that the hops traversed to a destination are, in fact, the shortest path to the destination.

We believe we can embed a latency graph in a hyperbolic space and define UrDHT such that it operates within this space [15] [16].. The end result would be a DHT with latency embedded into the overlay. Nodes would respond to changes in latency and the network by rejoining the network at new positions. This approach would maintain the decentralized strengths of DHTs, while reducing overall delay and communication costs.

References

[1] "Redis," http://redis.io.

[2] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies*. Springer, 2001, pp. 46–66.

[3] B. Cohen, "Incentives build robustness in bittorrent," in *Workshop on Economics of Peer-to-Peer systems*, vol. 6, 2003, pp. 68–72.

[4] B. Benshoof, A. Rosen, A. G. Bourgeois, and R. W. Harrison, "A distributed greedy heuristic for computing voronoi tessellations with applications towards peer-to-peer networks," in *Dependable Parallel, Distributed and Network-Centric Systems, 20th IEEE Workshop on.*

[5] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*. ACM, 2000, pp. 163–170.

[6] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*. Springer, 2002, pp. 53–65.

[7] S. Fortune, "A sweepline algorithm for voronoi diagrams," *Algorithmica*, vol. 2, no. 1-4, pp. 153–174, 1987.

[8] D. F. Watson, "Computing the n-dimensional delaunay tessellation with application to voronoi polytopes," *The computer journal*, vol. 24, no. 2, pp. 167–172, 1981.

[9] M. Bern, D. Eppstein, and F. Yao, "The expected extremes in a delaunay triangulation," *International Journal of Computational Geometry & Applications*, vol. 1, no. 01, pp. 79–91, 1991.

[10] A. Rosen, B. Benshoof, R. W. Harrison, and A. G. Bourgeois, "Urdht," https://github.com/UrDHT/.

[11] G. S. Manku, M. Bawa, P. Raghavan *et al.*, "Symphony: Distributed Hashing in a Small World." in *USENIX Symposium on Internet Technologies and Systems*, 2003, p. 10.

[12] J. M. Kleinberg, "Navigation in a small world," *Nature*, vol. 406, no. 6798, pp. 845–845, 2000.

[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 149–160, August 2001. [Online]. Available: http://doi.acm.org/10.1145/964723.383071

[14] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 775–787.

[15] R. Kleinberg, "Geographic routing using hyperbolic space," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*. IEEE, 2007, pp. 1902–1909.

[16] A. Cvetkovski and M. Crovella, "Hyperbolic embedding and routing for dynamic graphs," in *INFOCOM 2009, IEEE*. IEEE, 2009, pp. 1647–1655.

[17] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, pp. 41–53, 2004.

[18] O. Beaumont, A.-M. Kermarrec, and É. Rivière, "Peer to peer multidimensional overlays: Approximating complex structures," in *Principles of Distributed Systems*. Springer, 2007, pp. 315–328.

[19] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek, "Comparing the performance of distributed hash tables under churn," in *Peer-to-Peer Systems III*. Springer, 2005, pp. 87–99.